A FEASIBILITY ASSESSMENT OF A FINITE ELEMENT REAL TIME, TIME OPTIMAL CONTROLLER

by

DONALD ALLAN SMITH

B.S., Kansas State University, 1985

A THESIS

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

MECHANICAL ENGINEERING

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

J. Garth Thompson

Major Professor

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS iv

LIST OF TABLES vi

ACKNOWLEDCEMENTS vii

Chapter

1. INTRODUCTION 1

   The development of Minimum-Time Control 1
   Previous Work 2
   Project Overview 7

2. GENERATION OF THE FINITE ELEMENT FINAL TIME GRID 12

   The Square Grid 12
   The R-Theta Grid 13
      The Crid Construction 13
      The Grid Formulation 14
      Isochrone Solution 20
      Finite Element Grid Verification 24
   Cenerating the Control 25
      The Square Grid 25
      R-Theta Grid 28

3. COMPUTER TEST SYSTEM 31

   The Overall System 31
   Hardware Details 34
      The Two PC's 34
      Communication Hardware 35
   Software Details 35

4. PERFORMANCE OF THE MINIMUM TIME CONTROLLER 39

   Testing the Controller 39
   Cenerating control from the square grid 43
   Cenerating control from the R-Theta grid 45

5. CONCLUSIONS AND RECOMMENDATIONS 78

Conclusions of the Investigation 78
Recommendations for Further Study 79

APPENDICES

1. THE COMMUNICATION HARDWARE 81

The PDMA-16 and DMA Hardware 81
The Mechanics of the Communication 85
The Handshaking Hardware 87
Parts List 94

2. THE COMPUTER TEST SYSTEM SOFTWARE 102

ZSIMUL and ZRSIMUL 102
HPSIMUL - Switching Curves Control 126
HPSIMUL - Finite Elements Control 133
Principle Hardware Configuration Subroutines 154
Assembly Support Subroutines 172

REFERENCES 194

LIST OF ILLUSTRATIONS

1.1	Isochrone plot for a double integrator system	11
2.1	Node numbering for the R-Theta grid	15
2.2	Element numbering for the R-Theta grid	16
2.3	Finite Element Grid Performance	27
2.4	Element types for the square grid	30
2.5	Element types for the R-Theta grid	30
3.1	The computer test system	32
4.1	Pseudo-Random number set	42
4.2	Distribution of Pseudo-Random number generator	42
4.3	Isochrone plot for the square grid	44
4.4.a	History plot for square grid control, set 1	46
4.4.b	History plot for square grid control, set 2	47
4.5.a	Element plot for grid 1	52
4.5.b	Isochrone plot for grid 1	53
4.5.c	History plot for grid 1 control, set 1	54
4.5.d	History plot for grid 1 control, set 2	55
4.6.a	Element plot for grid 2	56
4.6.b	Isochrone plot for grid 2	57
4.6.c	History plot for grid 2 control, set 1	58
4.6.d	History plot for grid 2 control, set 2	59
4.7.a	Element plot for grid 3	60

4.7.b Isochrone plot for grid 3 61

4.7.c History plot for grid 3 control, set 1 62

4.7.d History plot for grid 3 control, set 2 63

4.8.a Element plot for grid 4 64

4.8.b Isochrone plot for grid 4 65

4.8.c History plot for grid 4 control, set 1 66

4.8.d History plot for grid 4 control, set 2 67

4.9.a Element plot for grid 5 68

4.9.b Isochrone plot for grid 5 69

4.9.c History plot for grid 5 control, set 1 70

4.9.d History plot for grid 5 control, set 2 71

4.10.a Element plot for grid 6 72

4.10.b Isochrone plot for grid 6 73

4.10.c History plot for grid 6 control, set 1 74

4.10.d History plot for grid 6 control, set 2 75

4.11.a Element plot for grid 7 76

4.11.b Isochrone plot for grid 7 77

A1.1 Symbol definitions for circuit diagrams 96

A1.2 Circuit diagram of the PDMA interface 97

A1.3 Circuit diagram of transfer request frequency generator 98

A1.4 Circuit diagram of transfer request controller 99

A1.5 Circuit diagram of interrupt frequency filter 100

A1.6 Circuit board layout 101

LIST OF TABLES

4.1 Grid characteristics 51

4.2 Controller performance 51

A1.1 Logic diagram of data bus enable 88

A1.2 Pin out for the 37 and the 40 connectors 90

A1.3 Zenith transfer request logic table 91

A1.4 Hewlett Packard transfer request logic table 92

ACKNOWLEDGEMENTS

I would like to thank the people which have contributed to this thesis. Thanks go to my graduate committee for their time and support, Dr. J. Garth Thompson, Dr. Chi-Lung Huang, and Dr. Donald Lenhert. For my major advisor, special thanks go to Dr. Warren White Jr. for his support, direction and the confidence throughout the work. The Department of Mechanical Engineering, Kansas State University for the equipment and financial support. Dr. Tom Gallagher, Director of the Computing and Telecomunications Center, David Naas, and the rest of the Center's staff for the help and support. My parents for all the support and advice.

Chapter 1

INTRODUCTION

The Development of Minimum-Time Control

Automatic control systems are used in hundreds of applications. They regulate many aspects of daily life. Simple systems to control the temperature in a house to advanced systems to control a satellite are all part of the technological world. A major part of the control systems in use are based upon a proportional-integral-derivative (PID) control. These PID controls systems are often based upon traditional performance criteria in both the time and frequency domain such as rise time, settling time, peak overshoot, gain margin, and phase margin. These performance criteria are sometimes not enough to achieve the desired design goals and another technique such as optimal control is used.

Optimal control seeks to minimize (or maximize) a system performance criteria, such as time, fuel, or efficiency. Minimum time control seeks to move a system, described by the relation

$$\dot{\underline{X}}(t) = a(\ \underline{X}(t),\ \underline{U}(t)\ ) \qquad (1.1)$$

from some initial position or state to some final configuration in the least amount of time. For this system, $\underline{X}$ is the n dimensional state

vector, $\dot{\underline{X}}$ is the rate of change of the state vector, and $\underline{U}$ is the control driving the system.

Optimal controls are used in a variety of applications. Manufacturing operations use optimal control techniques to minimize production costs. Optimal control techniques are used throughout the plant on robotic and machining equipment. More specific applications of optimal controls are used by the military on missile tracking systems, guidance systems, and various other weapons systems. In outer space, the jet controls of a satellite are an on/off system, thus bang-bang controls. The applications of optimal controls continues to grow and expand.

Previous Work

The technological change along with the new demands on industry in the 1950's helped motivate the development of optimal control theory. The mathematical development of optimal control theory was led by Bushaw [1] and Bellman [2]. The Russian mathematician, Pontryagin [3] published his minimum-maximum principle, "The optimal control to obtain minimum-time response is maximum effort throughout the interval of operation." This defined controls operating at their limits or "bang-bang" controls. These theories were proved by LaSalle [4] and various others. A paper by Oldenburger and Thompson [5] was one of the first attempts to synthesize optimal controls for design purposes. Methods were presented for finding the switching functions in terms of the state variables for a variety of

second and third order systems. The early problems treated were simple enough to be solved analytically. With the advent of the computer, techniques for solving the more complicated problems were developed. These new approaches were mainly numerical techniques.

There are four areas of numerical techniques to solve the minimum-time problem: 1) minimization of the discretized problem, 2) iteration on the initial or final values of the costate variables, 3) iteration on the switching time of the controls, and 4) generating control from the minimum-time isochrones.

Shetty [6] presented a finite element approach to solve for the optimal control of a two degree of freedom manipulator. Time is divided into uniform intervals and the state and costate variables are treated as unknowns at each time increment. The Hamilton-Jacobi equation is applied as a boundary condition at both the start and the end of the interval for which the control is sought. The method produced comparable results to those obtained by a continuous solution method. The method is sensitive to the closeness of the initial guess. Subrahmanyam [7] applied Newton's method to the time discretized problem. An interpolation polynomial was used to approximate the state and control history. A recursive formula is used to iterate until convergence.

The second technique used is iteration on the initial or final values of the costate variables. The basic technique iterates on the unknown initial values for the costate variables until convergence. Various techniques for updating the guesses have been used.

Knudsen [8] used a Newton-Raphson to iterate on the guesses for a single input, linear, time invariant system. Lastman [9] made guesses for the initial values of the costate variables and t_f. He then integrated the system forward in time. He used Newton's method to determine the switching times. Along the same path, Lasdon, Mitter, and Warren [10] used a conjugate gradient minimization to update the guesses. Lewine and Thorp [11] were also similar but used a second-variation decent technique to update the guesses. Kahn and Roth [12] used guesses for the final values of the costate vector for a model of a kinematic chain. The system was integrated backward in time to see if the initial state could be reached. The final values of the costate variables were iteratively changed until close approximations for the initial state were determined.

The third technique uses iteration on the switching times to solve the problem. Larson [13] presented a technique he called "time interval optimization." A Newton-Raphson iteration is used to adjust the switching times. A successive approximation procedure, the Piccard Method, was used to approximate the state variables. Smith [14] presented a technique for solving a linear system for which there are n switches. He arbitrarily chose a bang-bang control and then integrated forward in time to get the terminal error. The terminal error was used to improve the initial choice of the switching times by using two parallel optimization processes. Yastreboff [15] presented a technique also for linear systems. Control switching times are arbitrarily

selected. With the controls unbounded, the system is integrated forward in time to the desired terminal condition. The switching times are then adjusted to minimize the control magnitudes between each of the intervals and then the process is repeated. The solution is reached when the magnitude of the controls for all the intervals is the same. Davison and Monro [16] presented a technique for solving for the control for a non-linear time varying case. The number of switches for the system is selected. The system is then integrated forward in time. The switch times are varied and Rosenbrock's hill climbing method is used to iteratively adjust the switching times. Wen and Desrochers [17] presented a similar technique for solving for the switching times but used a gradient method. Niemann [18] improved upon the work of Wen and Desrochers. The original technique did not always yield a solution to achieve the final state.

It should be noted that the switching-time optimization techniques assume the system is bang-bang. The goal of the technique is to find a bang-bang control to achieve the final state. At times, such a solution is also time optimal.

The final technique covered is generating control from the system's minimum-time isochrones. Minimum-time isochronal surfaces are the set of states that can be taken to the final destination in the same minimum time, T^*. The minimum-time isochronal surfaces form a family of closed convex surfaces in the state space which expand monotonically from the final destination as T^* increases from zero to

infinity. Algebraic expressions for the isochronal surfaces for certain second order surfaces have been developed in Lee and Markus [19], Athans and Falb [20], and Ryan [21]. Explicit algebraic expressions have been derived for the isochronal surfaces for certain third-order systems with real eigenvalues and a single saturable control input by Ryan [22].

Rajendran [23] presented a technique using a finite element approach to solve for the minimum-time isochrones of the system. The system's state variables were treated as the independent variables therefore, time was eliminated from the analysis. The technique converges quickly and gives a reasonable approximation to the minimum-time isochrones and minimum time control.

Lee and Marcus [19] presented a technique suggested by Athans and Falb [20] for generating the control from a system's minimum-time isochronal surfaces. Luh and Shafran [24] presented an implementation of this technique for a fourth order linear system. The minimum-time isochrones were calculated for a discrete set of points using a known technique. The isochrones were approximated by a hyperellipsoidal function. The coefficients of the set of hyperellipsoidal functions are approximated by a set of continuous functions of state. The continuous functions were generated by a least squares fit to the calculated isochrone distributions. The approximate functions were then used to determine control. An implementation of the controller provided good results. The technique is limited by the density of the

original isochrone grid and the number of state points for the least squares fit.

A related paper by Smith [25] presents a technique for approximating the switching curves as linear-segments using a least-squares fitting technique. Points on the switching curves are generated by other known techniques. Linear segments are then fitted to the points. He applied this procedure to obtain an approximate expression for the switching surface of a triple integrator. The technique is similar to the minimum-time isochrones technique because the isochrones are approximated by a grid of discrete points.

Project Overview

This paper is an investigation into using the minimum-time isochrones to control a system. The system used is the double integrator problem,

$$\dot{\underline{X}} = A\underline{X} + BU \tag{1.2}$$

where

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1.3}$$

where X_i is the state variable, $\dot{X}_i$ is the rate of change of the state variable, and U is the control driving the system. The control U has the magnitude constraint of

$$|U| \leq 1 \ . \tag{1.4}$$

The cost function for the minimum time problem is

$$J = \int_0^{t_f} 1 \, dt \ . \tag{1.5}$$

Since the constraints on U are linear, the system is bang-bang. The optimal control Hamiltonian is

$$H(\underline{X}(t), U(t), \underline{\lambda}(t)) = 1 + \lambda_1(t)x_2(t) + \lambda_2(t)u(t), \tag{1.6}$$

where λ_i is the costate variable. To minimize the optimal control Hamiltonian, a control given by

$$u(t) = -\text{sgn}(\lambda_2(t)) \tag{1.7}$$

is chosen. The sgn is the signum function. From the optimal control Hamiltonian, the time derivatives of the costates are found to be

$$\dot{\lambda}_1(t) = 0, \tag{1.8.a}$$

and

$$\dot{\lambda}_2(t) = -\lambda_1(t). \tag{1.8.b}$$

For this particular formulation of the minimum time double integrator problem, Bryson and Ho [26] demonstrate a continuous dependence of the costates on the final time as

$$\lambda_1 = \frac{\partial t_f}{\partial x_1} , \tag{1.9.a}$$

and

$$\lambda_2 = \frac{\partial t_f}{\partial x_2} . \tag{1.9.b}$$

Note that λ_1 and λ_2 point in the direction of greatest increasing final time. From Rajendran's [23] discrete finite element work

$$u = \operatorname{sgn}(\lambda_2(1-x_2\lambda_1)), \tag{1.10}$$

the discrete form of control. The system's minimum-time isochrones can be used to calculate λ_1 and λ_2 from which the control is then calculated. Figure 1.1 shows a plot of the isochrones of a double integrator system.

Chapter Two develops the two discrete minimum-time isochrone grids used. The first grid was from Rajendran's [23] development using a square finite element grid. The second grid is developed using a R-Theta grid based upon the switching curves.

Chapter Three describes the computer test system used to simulate the control system. The test system consisted of 2 personal computers, one a digital controller and the other simulates a double integrator system. Communication was handled by using a high speed, parallel communication link based upon Direct Memory Access (DMA).

Chapter Four presents the results of the simulations. Control generated by minimum-time isochrones is compared to control generated using the switching curves. Two sample rates were used, $\Delta t = 0.02$ and 0.03 seconds.

Chapter Five presents the conclusions of the investigation and gives recommendations for further work.

INTEL is a registered trademark of the INTEL Corporation. Hewlett Packard Vectra is a registered trademark of Hewlett Packard Corporation. Zenith Z-159 is a registered trademark of Zenith Corporations. MetraByte PDMA-16 is a registered trademark of MetraByte Corporation.

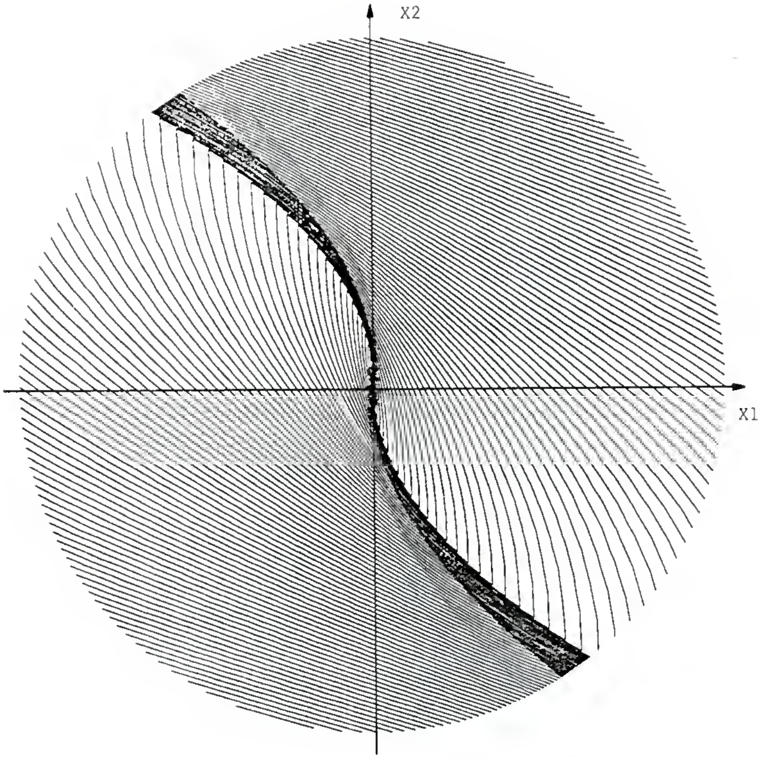


Figure 1.1: Isochrone plot for a double integrator system

Chapter 2

GENERATION OF THE FINITE ELEMENT FINAL TIME GRID

The Square Grid

The use of the square grid to generate the finite element final time grid is a direct implementation of Rajendran's [23] work . The work was also presented in a paper by White and Rajendran [27]. In their development it was shown that

$$u = \text{sgn}[\lambda_2(1-x_2\lambda_1)] \tag{2.1}$$

for their discrete finite element work. Substituting Equations (1.9.a) and (1.9.b) for λ_1 and λ_2 gives the continuum function for control of the system from the final time grid.

For the square grid approximation, the switching curve passes through the elements. The switching curve is approximated by a horizontal straight line across the element, parallel to the X_1 axis. This approximation causes discontinuity of the switching curve between elements. Using this straight line approximation causes the system to chatter about the straight line. The chattering increases as the state origin is approached because the gradient of the switching curve is also increasing in magnitude.

A new grid is needed to provide a smoother control. A denser placement of nodes around the state origin would give a better approximation of the control as the gradient of the switching curve increases. Polar geometry would provide a better tool to satisfy these constraints.

The R-Theta Crid

The Grid Construction

The polar geometry grid will be based upon the switching curve of the system. The switching curves will be approximated by the sides of a string of elements (see nodes 1, 2, 8, 14, and 20 in Figure 2.1). This is a linear segment approximation to the switching curve. The number of points along the switching curve is a user input variable, # CIRCLES. Each discrete point on the switching curve will be rotated about the origin to generate a series of nodes. The angular displacement of each node will be a function of an input variable. The input variable will be the total number of rotations, # SPOKES. This input number must be an even number to insure the symmetry of the system and the two switching curves. To increase the number of elements around the state origin, a user input grid scaling factor will be used, CRID FACTOR. The radius of the current circle will be a function of the previous circle, and the CRID FACTOR,

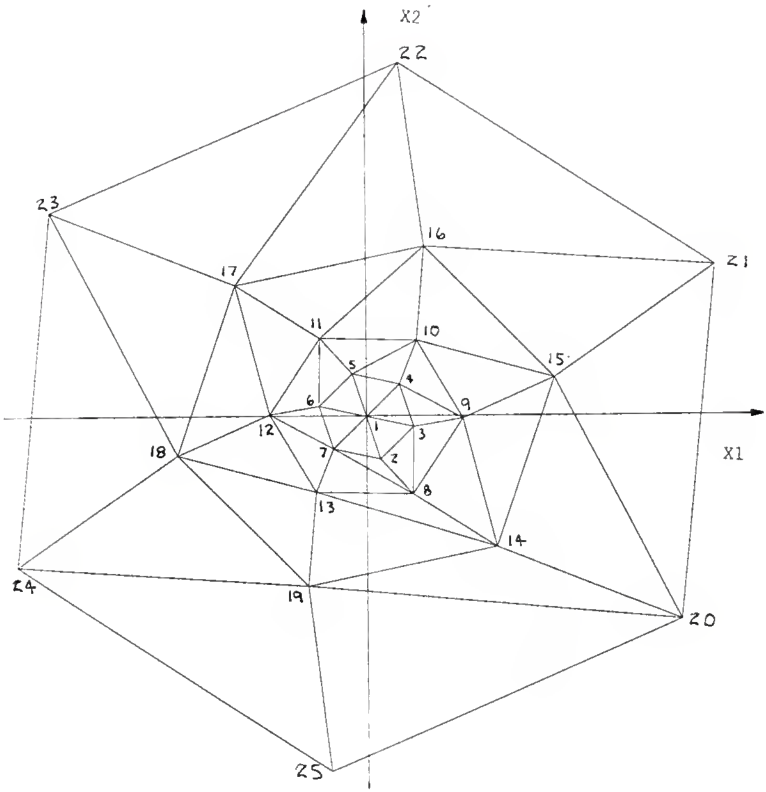
$$r_{i+1} = r_i * \text{CRID FACTOR} . \qquad (2.2)$$

The user specifies the total scale size, SCALE which is the radius of the outer circle. SCALE is then combined with the number of circles and the grid factor to determine the radius of the inner circle.

The node numbering for a simple grid is presented in Figure 2.1. The element numbering is shown for the same grid parameters in Figure 2.2. Note the change in element styles along the row of elements on the lower side of the switching curve in the fourth quadrant of Figure 2.2. This was implemented to keep the size of the bandwidth of the system to a minimum.

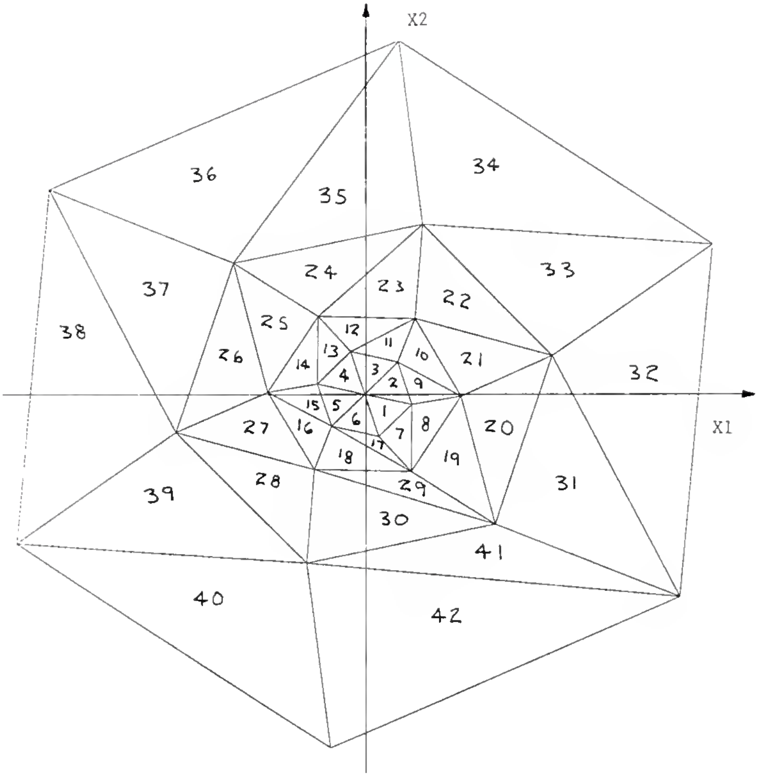
The Grid Formulation

While it was the purpose of this investigation to determine if the finite element isochrone distribution could be used to provide suitable control, some opportunity existed for testing different finite element formulations so as to assess the ability of the formulation to provide accurate isochrone information. In a previous study ite and Rajendran [27] used the Hamilton-Jacobi equation coupled with the minimum time functional to determine not only the isochrone values but also the control with an iterative solution process. The solution produced by this method is not suitable for control purposes owing to the large amounts of chatter as mentioned earlier. Because of this chatter a new finite element grid based upon the switching curve was chose for this work. The new grid eliminates the necessity of iteratively determining the control since the problem is now linear. Since the finite element grid can be built and the control determined in advance by integrating the state equations backward in time, the question arose as to what freedom does this introduce in determining



spokes	=	6
circles	=	4
scale	=	5.0
grid factor	=	2.0

Figure 2.1: Node numbering for the R-Theta grid



spokes	=	6
circles	=	4
scale	=	5.0
grid factor	=	2.0

Figure 2.2: Element numbering for the R-Theta grid

the isochrones. Specifically, if the control distribution is known is the Hamilton-Jacobi equation sufficient to determine the isochrones? A related question consists of finding the necessary boundary conditions if this approach should prove possible.

To be presented are the steps leading to a pure Hamilton-Jacobi formulation, the results of this approach, and the method selected to provide a solution for the purposes of this investigation.

The isochrone distribution satisfies the first order partial differential equation

$$\underline{\nabla} t_f \cdot \dot{\underline{X}} = -1 \tag{2.3}$$

where the $\underline{\nabla}$ is the vector gradient operator given by

$$\underline{\nabla}^T = \left(\frac{\partial}{\partial x_1} \ \frac{\partial}{\partial x_2} \ \cdots \ \frac{\partial}{\partial x_n} \right) \tag{2.4}$$

for a system of order n. Equation 2.3 has many different solutions depending upon the final state. Another way to describe Equation 2.3 is that the optimal control Hamiltonian is a constant of the motion which can be stated on a continuum basis as

$$\frac{d}{dt} H = \frac{d}{dt} (1+\underline{\lambda}\cdot\dot{\underline{X}}) = -\dot{\underline{X}}\cdot\underline{\nabla}(\dot{\underline{X}}\cdot\underline{\nabla} t_f) = 0 \tag{2.5}$$

where the time derivative translates into the spatial operator

$$\frac{d}{dt} = -\dot{\underline{X}}\cdot\underline{\nabla} \ . \tag{2.6}$$

Equation (2.5) provides a second order, partial differential equation for the final time distribution and is well suited for finite element analysis. In order to develop a finite element formulation of Equation (2.5) we will use the Galerkin method, as described by Huebren and Thorton [28]. Assume the state variables are described by the interpolation function given by

$$t_f(\underline{X}) = \sum_{i=1}^{3} N_i(\underline{X})\ t_{f_i} = \{N\}^T\{t_f\}\ . \tag{2.7}$$

The functions N_i are linear finite element interpolation polynomials for a two dimensional domain and are the same interpolation polynomials used by White and Rajendran [28].

The Galerkin method when applied to Equation (2.5) produces

$$-\int_{D_e} \{N\}\ \dot{\underline{X}}\cdot\underline{\nabla}(\dot{\underline{X}}\cdot\underline{\nabla}t_f)dD_e = 0 \tag{2.8}$$

where D_e is the domain of the element. Integrating Equation (2.8) by parts produces

$$-\int_{S_e} \{N\}(\dot{\underline{X}}\cdot\underline{S})(\underline{X}\cdot\underline{\nabla}t_f)dS_e + \int_{D_e} (\underline{\nabla}\{N\}\cdot\dot{\underline{X}})(\dot{\underline{X}}\cdot\underline{\nabla}t_f)dD_e +$$

$$\int_{D_e} \{N\}(\underline{\nabla}\cdot\dot{\underline{X}})(\dot{\underline{X}}\cdot\underline{\nabla}t_f)dD_e = 0 \tag{2.9}$$

where S_e is the exterior surface of the element and $\underline{S}$ is the outward unit normal to the surface. By invoking the Hamilton-Jacobi equation in the first and last terms of Equation (2.9) we have

$$\int_{D_e} (\underline{\nabla}\{N\}\cdot\underline{\dot{X}})(\underline{\dot{X}}\cdot\underline{\nabla}t_f)dD_e = \int_{D_e} (\{N\}\underline{\nabla}\cdot\underline{\dot{X}})dD_e - \int_{S_e} \{N\}(\underline{\dot{X}}\cdot\underline{S})dS_e \quad (2.10)$$

Equation (2.10) has a symmetrical element matrix and the boundary integral needs to be computed only on the exterior boundary of the problem since the boundary integral cancels on inter-element boundaries.

The divergence of $\underline{\dot{X}}$ in the right hand side of Equation (2.10) requires some examination. In areas where the control is constant the divergence of $\underline{\dot{X}}$ vanishes. Where the control is changing the divergence of $\underline{\dot{X}}$ will consist of impulses which when integrated will provide a contribution to the element. These contributions will occur at nodes which are located along the switching curve. This introduces two possible solution methods. The first is to leave the node on the switching curve unspecified and include the nodal contribution caused by the discontinuity of control. The second is to specify the value of the final time at the node since this information is available from the grid construction. Specifying the final time at the node will eliminate the need for this particular element contribution.

An element assembly and solution procedure based upon Equation (2.10) was developed which used the polar grid described earlier. For the boundary conditions the user had the option of either

1) specifying the final time at the origin, 2) specifying the final time at the origin and along the switching curve, or 3) specifying the final time at the origin, along the switching curve, and on the external boundary.

In all boundary condition cases just described, no satisfactory solutions could be obtained by this technique. Away from the switching curve the isochrones tended to become horizontal and resembled a solution found by White and Rajendran [28]. At this point the investigation into possible finite element formulations was abandoned. The conclusion drawn from this brief examination is that Hamilton-Jacobi together with the particular finite element interpolation used is not sufficient to determine the minimum time isochrones.

Isochrone Solution

The starting point for the determination of the minimum time isochrones is the least squares finite element formulation of White and Rajendran [27]. In this work there is no longer a need to iterate in order to solve for the equations owing to the method chosen to construct the grid. To be presented is the least squares finite element performance index, the finite element equations, and the boundary conditions chosen to provide a solution for this investigation.

The least squares, finite element performance index is given by

$$I_{fe} = \int \frac{1}{2} [(1+\dot{\underline{X}}\cdot\underline{\nabla}t_f)^2 + K(t_f-(2x_1\ x_2)\cdot\underline{\nabla}t_f)^2]dA \qquad (2.11)$$

where K is a constant of unit magnitude which insures consistency of dimensional units. Equation (2.11) is based upon two separate equations. The first is the Hamilton-Jacobi equation given as Equation (2.3). The second equation, given by

$$t_f = (2x_1\ x_2)\cdot\underline{\nabla}t_f \qquad (2.12)$$

is exactly the minimum time functional. Equation (2.12) is obtained by repeatedly integrating the minimum time functional by parts. The solution obtained by minimizing Equation (2.11) is the best fit to the Hamilton-Jacobi equation and the minimum time functional.

The interpolation of the final time over an element is provided by

$$t_f = \{N\}^T\{t_f\} \qquad (2.13)$$

where $\{t_f\}$ is the vector of final time values at each element vertex. The ith element of the vector $\{N\}$ is given by

$$N_i(x_1,x_2) = (a_i + b_i x_1 + c_i x_2)/2\Delta \qquad (2.14)$$

where a_i, b_i, and c_i are all constants determined by the element geometry and the Δ is the area of the triangle. The function N_i has the property of being unity at node i and zero at the remaining nodes. Using Equation (2.13), the gradient of the final time becomes

$$\underline{\nabla} t_f = \underline{\nabla}\{N\}^T\{t_f\} = \frac{1}{2\Delta} \begin{bmatrix} b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix} \{t_f\} \quad . \tag{2.15}$$

The interpolation given by Equation (2.13) is substituted into the performance index of Equation (2.11). The finite element equations are obtained by minimizing Equation (2.11) through

$$\frac{\partial}{\partial\{t_f\}} I_{fe} = 0 \quad . \tag{2.16}$$

Performing the operation indicated by Equation (2.16) produces after considerable algebra, the matrix equation

$$[ELM]\{t_f\} = \{ERSV\} \tag{2.17}$$

where

$$\begin{aligned} ELM_{ij} = \frac{1}{4} (& H_{11}b_i b_j + H_{12}(b_i c_j + b_j c_i) + H_{22}c_i c_j)/\Delta^2 \\ & - [(2x_{1_i} + 6x_{1_{AVE}})b_j + (x_{2_i} + 3x_{2_{AVE}})c_j]/24 \\ & - [(2x_{1_j} + 6x_{1_{AVE}})b_i + (x_{2_j} + 3x_{2_{AVE}})c_i]/24 \\ & + (1 + \delta_{ij})\Delta/12 \end{aligned} \tag{2.18}$$

and

$$ERSV_i = - \frac{1}{2} (b_i \cdot x_{2_{AVE}} + c_i \cdot u) \quad . \tag{2.19}$$

In Equations (2.18) - (2.19) the quantities $x_{1_{AVE}}$ and $x_{2_{AVE}}$ are the centroidal values of the state variables in the element, u is the control in the element, and δ_{ij} is the Kronecker delta. The quantities H_{11}, H_{12}, and H_{22} are given by

$$H_{11} = \int_A (x_2^2 + 4x_1^2)\,dA, \tag{2.20}$$

$$H_{12} = \int_A (ux_2 + 2x_1x_2)\,dA, \tag{2.21}$$

and

$$H_{22} = \int_A (1 + x_2^2)\,dA \; . \tag{2.22}$$

Equations (2.20) - (2.22) are easily evaluated as moments of the triangular area about the state axes.

Equation (2.17) for each element is evaluated and assembled together to form the global system of equations in the standard finite element fashion (see Huebren and Thorton [28]). The boundary conditions used to complete the formulation were to specify the final time at the state origin, along the switching curve, and at the external boundary. The specification of the final time at the state origin and the external boundary is a necessary step as pointed out by White and Rajendran [27]. The specification of the final time along the switching curve is not necessary to produce a correct solution however, since this information was determined independently when

building the grid and it was desired to have accurate final time gradients in the vicinity of the switching curve, the final time at each switching curve node was specified.

Finite Element Grid Verification

The results of the finite element grid will be tested by two techniques: 1) the performance index versus the number of nodes, and 2) the error from the true solution versus the number of nodes.

If the formulation for the finite element grid is correct, the performance index given in Equation (2.11) should decrease monotonically with increasing number of nodes. With the increase in the number of nodes, the base ten logarithm of the number of nodes versus the base ten logarithm of the performance index is plotted in Figure 2.3. The data forms a straight line confirming the validity of the solution.

The second test will be a plot of the error from the true solution. The error is calculated by

$$\text{error} = \left[\frac{\int_A (t_{f_{ELEMENT}} - t_{f_{TRUE}})^2 dA}{\int_A t^2_{f_{TRUE}} \, dA} \right]^{\frac{1}{2}} \qquad (2.23)$$

The error should also decrease monotonically as the number of nodes is increased. Figure 2.3 shows a plot of the base ten logarithm of the number of nodes versus the base ten logarithm of the error. The data forms a straight line which confirms both the solution and the

convergence of the finite element results to the true solution as the size of the elements shrinks to zero.

Generating the Control

The Square Grid

The method the control subroutine uses to generate control has two parts: 1) determine the element number in which the current state position is located, and 2) calculate the control based upon that element.

The technique to determine which element corresponds to the current state position is simplified by using the square grid shape. Using the number of elements across the grid together with the element size, the element in which the current state is located can quickly be calculated.

The control calculation is a function of element type (see Figure 2.4). For an upper element

$$\lambda_1 = \frac{\partial t_f}{\partial x_1} = \frac{-(t_{f3}-t_{f1})}{L} \tag{2.24}$$

and

$$\lambda_2 = \frac{\partial t_f}{\partial x_2} = \frac{-(t_{f1}-t_{f2})}{L} \tag{2.25}$$

while for a lower element

$$\lambda_1 = \frac{\partial t_f}{\partial x_1} = \frac{-(t_{f3}-t_{f2})}{L} \tag{2.26}$$

and

$$\lambda_2 = \frac{\partial t_f}{\partial x_2} = \frac{-(t_{f1} - t_{f3})}{L} \qquad (2.27)$$

where t_{fi} is the final time value for node i and L is the length of the side of the element. Equation (2.1) is evaluated at each node. If the control is the same for all three nodes, then the element does not lie on the switching curve and the control at any node is valid. If the control has a different sign at any of the nodes, the current state position must be compared to the horizontal line approximation for the switching curve. The approximation line is located at the X_2 coordinate of

$$x_{2s} = \frac{1}{\lambda_1} \qquad (2.28)$$

and U_j = the control at the node having the unique sign. For an upper element, if x_2 is greater than x_{2s} then the control is opposite in sign to U_j whereas if this inequality is not satisfied then the control is equal to U_j. For a lower element, if x_2 is less than x_{2s} then the control is opposite in sign to U_j however should the state point be located on the opposite side of x_{2s} than the control is of the same sign as U_j.

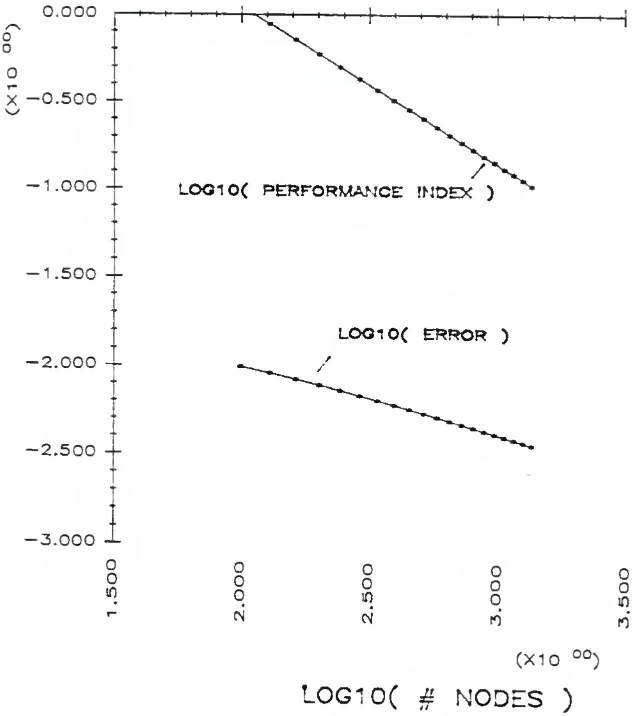


Figure 2.3: Finite element grid performance

The R-Theta Grid

The same main two steps are used by this subroutine as was used by the square grid algorithm.

The technique to determine the element number in which the current state position is located is difficult due to the grid shape. The first step is to determine the doughnut of the current state or the two circles between which the state is located. Four iterations of a bisection technique are first used to reduce the solution area. The initial values of the maximum and minimum radius are the outer radius and zero, respectively. The minimum radius is then increased until it is larger than the radius for the current state. The spoke number is determined by calculating the angle between the switching curve node in the fourth quadrant of the inner radius of the doughnut and the current state position. To determine if the position is in the inner or outer element, (see Figure 2.5) an imaginary line is drawn between the current state and the state origin. The intersection of this line and the element edge separating the upper and lower elements is calculated. If the distance to the intersection point is greater than the distance to the current state the element is an inner element. Failing this test it is an outer element. Two special cases must be accounted for which are: 1) if the current position is inside the inner-most radius, there is only a single element to each spoke, and 2) the last spoke - the row of elements in fourth quadrant under the switching curve form an approximate mirror image of the other shapes just above the switching curve.

The finite element grid presents a technique for solving for the system's isochrones. The control can now be calculated by using

$$u = \mathrm{sgn}\left(\frac{-\partial t_f}{\partial x_2} \right) . \tag{2.29}$$

The derivative $\frac{\partial t_f}{\partial x_2}$ is calculated by using Equation (2.15) with the area of the triangle calculated using Equation (2.14).

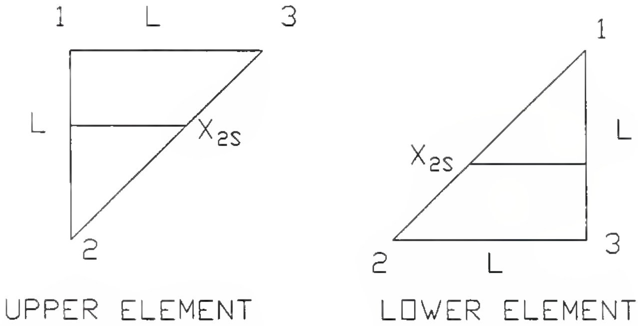


Figure 2.4: Element types for the square grid

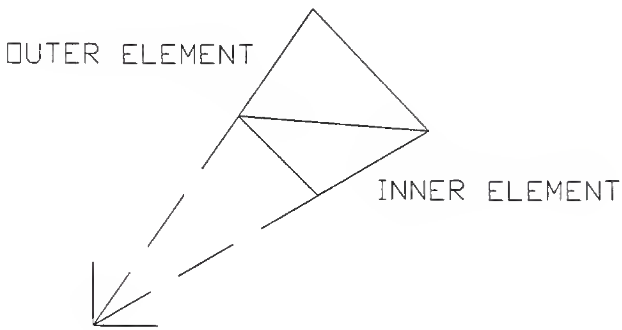


Figure 2.5: Element types for the R-Theta grid

Chapter 3

COMPUTER TEST SYSTEM

The Overall System

The computer test system is made up of two personal computers (PC's). One computer is the digital controller and the other simulates a double integrator system. A digital sample scheme is used to synchronize communications between the two computers. This communication is accomplished over a 16 bit parallel bus using Direct Memory Access (DMA). Figure 3.1 shows the system layout.

The task of the digital controller is to wait until the control system requests the control. When the request is made, the controller 1) receives the current state vector, 2) calculates the control for that state, and then 3) returns the control to the dynamic system. In a true plant-controller system, the sample rate would drive the digital controller. The controller samples the state of the plant and then specifies control for that instant of time.

The main function of the simulated control system begins at the start of a sample interval. When a sample interval is signaled, the dynamic system sends the current position to the controller. A short time later when the system receives the control from the controller it integrates the system forward in time using the new control.

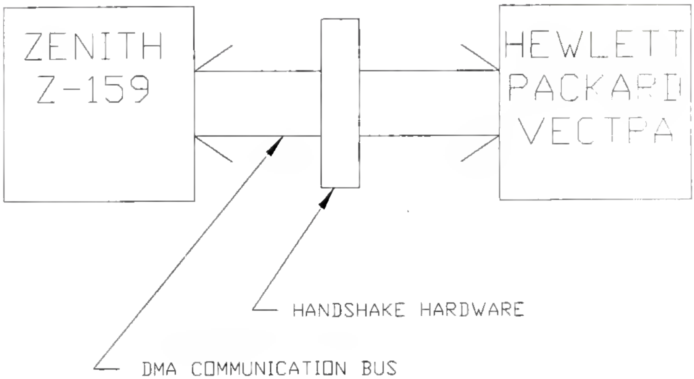


Figure 3.1: The computer test system

The DMA communication link operates at a rate of 50,000 16-bit words/second. The link can run up to approximately 100,000 words/second. This high speed communication link was used to minimize the time spent transferring data between the two computers and to relieve each processor from the communication overhead.

All control system tasks are interrupt driven. An interrupt is sent to the control system to signal a sample time interval. The control system initiates the DMA data transfer to send out the state vector. When the vector has been sent out, another interrupt signal is generated which sets a DMA transfer to receive the control. An interrupt is generated when the new control has been received. The system is integrated forward in time using the new control. The DMA transfer is then set up to send out the state vector at the next sample interrupt.

The controller's interrupt structure is similar. The DMA cycle is set up to receive the state vector. An interrupt is generated at the end of the transfer of the state vector. The interrupt routine calculates the control for that state position and then sets up the DMA transfer to send out the control. At the end of the transfer another interrupt is generated. This last interrupt then sets up the DMA transfer to receive the state vector of the next sample, finishing the cycle.

The interrupt driven structure is a very powerful system. This allows the central processor to be performing other tasks and only devoting attention to servicing the interrupt when absolutely

necessary. However, using the interrupt structure is disadvantageous in several ways. The software is more complicated. The development of software and hardware in the interrupt environment is extremely difficult. Using floating point arithmetic in the interrupt routines further complicates the problem. The state of the central processor has to be saved during the interrupt as well as the state of the numeric coprocessor. The interrupt handling problem turned out to be extremely difficult. The interrupt handling routines worked for a large part of the time but still occasionally failed. The reason for the occasional failures was not discovered, but was probably due to the floating point coprocessor.

Hardware Details

The Two PC's

The controller PC is a Hewlett Packard Vectra. The central processor is the INTEL 80286. The 80286 operates at a clock speed of 8 MHz. The HP Vectra has an INTEL 80287 numeric coprocessor. The HP Vectra is equipped with a 20 Mbyte hard drive, a 1.2 Mbyte floppy disk drive, a 360 Kbyte 5 1/4 inch floppy disk drive, a parallel port, and a serial port. DOS, version 3.10 was the operating system used.

The PC being used to simulate the double integrator system is a Zenith Z-159. The central processor is the INTEL 8086. It has a dual clock speed of 4.77/8 MHz (all the work was done at the 8 MHz setting). It has an INTEL 8087 numeric coprocessor. It is equipped with a 20 Mbyte hard drive, a 360 Kbyte 5 1/4 inch floppy disk drive, a parallel

port, and a serial port. DOS, version 3.20 was the operating system used.

Communication Hardware

Both PC's were equipped with a MetraByte PDMA-16 communication board. The PDMA-16 board is equipped with internal DMA and interrupt control hardware. The PDMA board also has an 8254 programmable interval timer. The PDMA board has the control hardware to transfer via DMA, 8 or 16 bit data from memory to its I/O ports or from its I/O ports to memory. The PDMA board has available for external use two input lines: 1) the DMA transfer request, and 2) interrupt request; 6 output lines: 1) DMA transfer acknowledge, 2) the 8254 timer, 3) the DMA transfer direction, and 4) three auxiliary lines; and 16 Input/Output data lines. The external lines are used for handshaking and data transfer. For a complete description see the PDMA-16 technical reference manual [29].

Special hardware was used to handle the handshaking between the PDMA-16 boards on each PC. This hardware is responsible for generating the sample time interrupts, the DMA transfer requests, and buffering the data bus. A detailed hardware description can be found in the Appendix 1.

Software Details

The two main types of software used are: 1) software used for hardware configuration, and 2) software used for the simulation. The software was written and compiled with 3 languages: 1) Microsoft C V3.00, 2) Microsoft Fortran77 V3.31, and 3) Microsoft Macro Assembler

V4.00. Microsoft Linker V3.05 was used for linking the object modules and libraries. All the compilation and linking was done on the Zenith running under DOS V3.20. Most of the software was written in C. The language is very efficient and can be used as a high or low level language. Assembly language was used for subroutines that either required speed, had high usage, or serviced interrupts. All the assembly subroutines were written in a C callable format. The development of the finite element grid was in Fortran. Rather than rewriting the Fortran subroutines, they were interfaced with the C language controller program. The use of all Microsoft software made the interface of the different languages a straight-forward task.

The software used for hardware configuration was modelled after MetraByte's PDMA software. MetraBytes's software consisted of assembly code in a BASIC language callable format. This format was sufficiently different from that required by the C language that implementing their software would have required a major revision of the software. The major hardware configuration routines are titled MODE#, where # is either 0,1,3,7, or 8. These subroutines perform the following tasks:

0) Initialization of the hardware

1) DMA transfer set up

3) Set the output rate of the interval timer

7) Set up interrupts - Microsoft C small model format

8) Set up interrupts - Microsoft C large model format

The software for the simulation is broken up into two major blocks. These are:

1) the control system - Zenith Z-159

2) the controller - Hewlett Packard Vectra.

The control system software, ZSIMUL is used to simulate a double integrator system. Provided there is no change in the handshaking procedure, ZSIMUL remains the same for all developments of the controller. ZSIMUL initializes the hardware, initializes the starting position of the control system, and begins the simulation. The path history is saved throughout the simulation. When the termination criteria is met, ZSIMUL restablishes the environment, writes the history to disk, and exits. ZSIMUL looks for a user requested exit while running the simulation. The simulation is gracefully exited by pressing the 's' key. A graceful exit uses the program's interrupt installation programs to return the interrupt structure to its default state. There is a version of ZSIMUL called ZRSIMUL that generates a pseudo-random number for the starting position. ZRSIMUL is used to simulate a large number of starting points for the controller so that a statistical assessment can be made of the controller performance.

The controller software, HPSIMUL initializes the hardware and provides the control for the control system. HPSIMUL has no user visible output other than starting and ending. The routine is in a continuous loop waiting for ZSIMUL to ask for control. Stopping the program is accomplished in the same manner as it was for ZSIMUL. HPSIMUL looks for a 's' key press to exit gracefully. The routine runs in an infinite loop so that ZRSIMUL and ZSIMUL would both work with HPSIMUL.

For a complete discussion of the hardware and the communication process see Appendix 1. A discussion of the software and a listing of the source code is provided in Appendix 2.

Chapter 4

PERFORMANCE OF THE MINIMUM TIME CONTROLLER

Testing the Controller

In Chapter One the basis of this investigation was presented. The double integrator control system and a technique for generating the control for the system were given. Chapter Two developed the technique used to generate the final time grid which is used to calculate control. Two types of grids were presented, a square grid, and an R-Theta grid. Chapter Three presented the basic design of the computer simulation system used. A personal computer was used to simulate a double integrator system. Another personal computer was used as a digital controller. A DMA communication link was used to exchange information between the two computers.

The simulations were conducted for two different sample rates, $\Delta t = 0.02$ and 0.03 seconds. Simulations were performed using each of the finite element final time grids. The square grid did not produce very fine control and thus was not extensively tested. The R-Theta grid produced favorable results. Various grid densities were generated and used to control the double integrator system. For each case, a plot of the path history is presented for four starting positions. The four positions are (+5,+5), (-5,-5), (+5,-1), and (-5,+1). The first

two points will be referred to as "set 1" while the last two points will comprise "set 2." The positions are used in mirrored pairs as a check since the results should be symmetrical about the origin.

The R-Theta grid was tested on 500 different starting points. The 500 points were generated using the pseudo-random number generator included with Microsoft C, V3.00 library denoted below as rand(). The default seed was used as the starting point. The pseudo-random number generator produces a integer number from 0 to 32,767. The initial position range of X1 and X2 is ±6.0. This range is sufficient to keep the resulting phase trajectory of X1 and X2 within a range of ±28.50. This is used to control the size of the grid required. The algorithm to generate X1 and X2 is

```
1.   X1 = rand()/5461.1667
2.   X2 = rand()/5461.1667
3.   SignX1 = rand()
4.   SignX2 = rand()
5.   if signX1 > 16383 then X1 = -X1
6.   if signX2 > 16383 then X2 = -X2
```

This algorithm provides a simple, repetitive way to generate a set of numbers. Figure 4.1 shows the set of numbers generated.

To test the pseudo-random number generator, 1.6E+7 numbers were generated. A grid was set up in the ±6.0 working space. Each element in the grid had dimensions of 0.30 X 0.30, this produced a total of 1,600 elements. Counters were then set up for each element. The 1.6E+7 points were placed into the corresponding elements on the grid. For each point inside an element, the counter for that element was incremented by one. For a perfect random number generator, there

should be 10,000 points inside the boundaries of each element. Figure 4.2 shows a plot of the counters for the 1,600 elements. The bandwidth about the 10,000 points-line is acceptable, thus the pseudo-random number generator is close to a true random number generator.

The simulation is stopped when the path enters into a circle about the state origin of radius = 0.06. This is 1% of the valid starting range and 0.21% of the total possible path range. The stopping radius was chosen large enough to keep the oscillation about the final destination to a minimum. If too small a range is used, several switches are required for the path to reach the stopping criterion. The exact time the path crossed the stopping circle is calculated by finding the intersection between the stopping radius circle and a parabolic curve fitted between the previous position and the position inside the stopping circle.

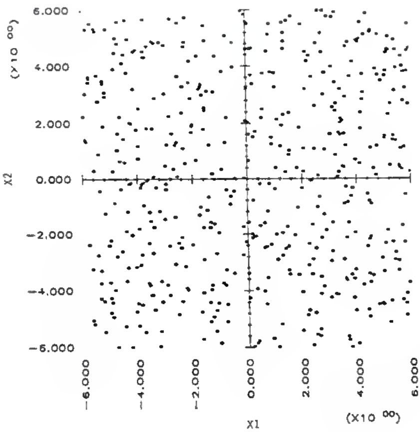


Figure 4 1: Pseudo-Random number set

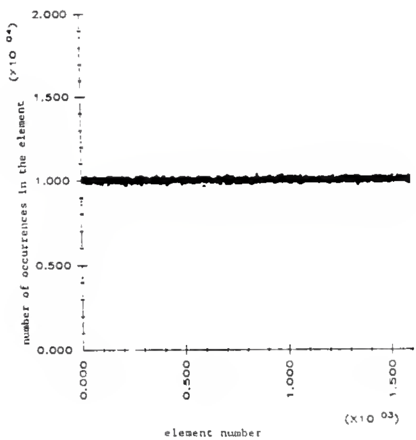
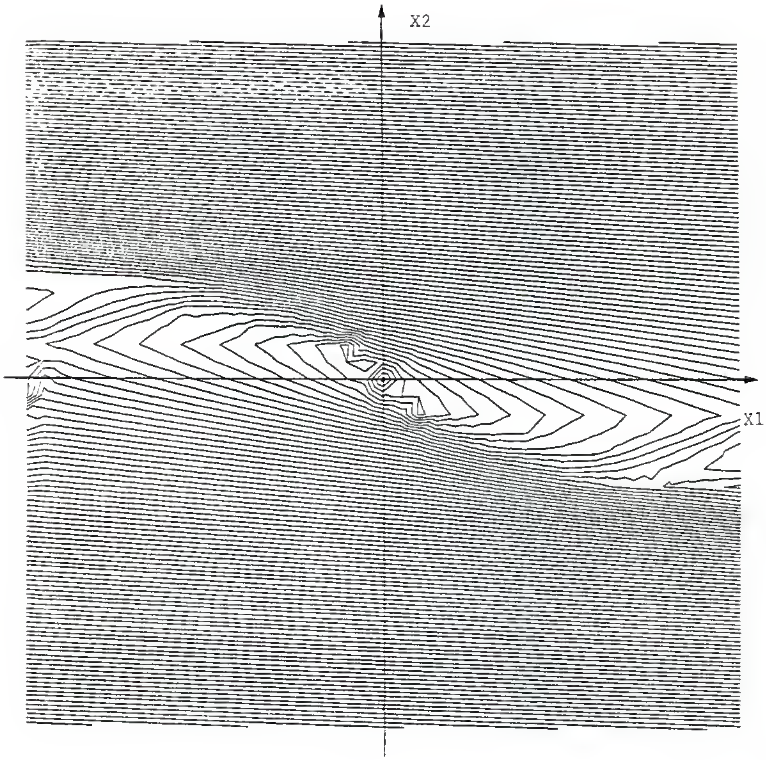


Figure 4 2 Distribution of Pseudo-Random number generator

A slight error is introduced by the fourth order Runga-Kutta integration. The integration scheme has a tendency to shift the path towards the focus of the parabola. This causes the path to reach the stopping circle sooner than the theoretical path would have reached it. The effect is minimal and only occasionally causes a better-than-optimal time to be achieved.

Generating Control From the Square Grid

The square finite element final time grid used has the dimensions of X1 = ±20.0 and X2 = ±20.0. Each element has dimensions of 1.0 X 1.0 for the base and height. Using these grid dimensions, there are 80 X 80 elements, and 41 X 41 nodes. This is a coarse grid but provides an indication of how well it can be used to generate control. Figure 4.3 shows an isochronel plot for this square grid. The isochrone approximation is fair in the doughnut region between the center and edges. The center region and the edges are not good approximations to the true isochrones. Since the control is based upon the gradient of the isochrones, this grid will not produce a reasonable control in these regions.



Square Grid

80 X 80 elements
element size = 1 X 1 unit

Figure 4.3: Isochrone plot for the square grid

Figures 4.4.a and 4.4.b show the path history for the four test points. The figures also show the ideal switching curve. The controller has a tendency to chatter in the vicinity of the switching curve. The control is poor near the exterior edges of the grid but closer to the state origin it is adequate. The control chatters about the straight line approximation of the switching curve in the element. The control is unsatisfactory using this finite element grid. A grid is needed that has a greater density of elements in the region of the final destination. The grid should also have larger elements near the grid exterior to reduce the storage space required.

Generating Control From the R-Theta Grid

The R-Theta grid, as discussed in Chapter Two is a grid built upon a radial coordinate system. The four grid shape determination factors are: 1) number of spokes (# spks), 2) number of circles (# circles), 3) radius - scale (scale), and 4) the grid factor (gridf). Seven different grids were generated and used by varying the four parameters. The first six grids provided reasonable control, the seventh did not. The seventh grid oscillated about the final destination and never spiraled into the stopping radius. The parameters used for the seven grids are listed along with the figure number for the path history for the four test points in Table 4.1. A plot of the finite element mesh for each grid is included. An isochronel plot for each grid is given to provide a rough estimate as to how well the grid will generate accurate control.

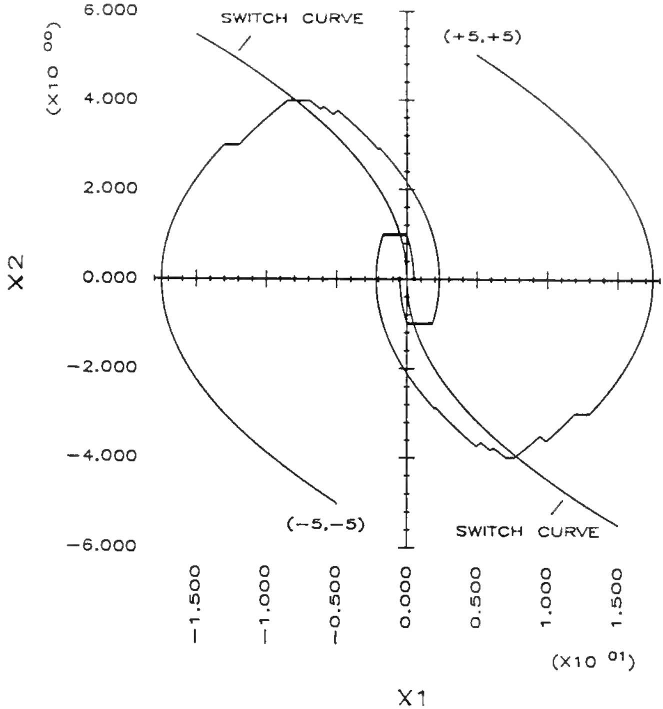


Figure 4.4.a: History plot for square grid control, set 1

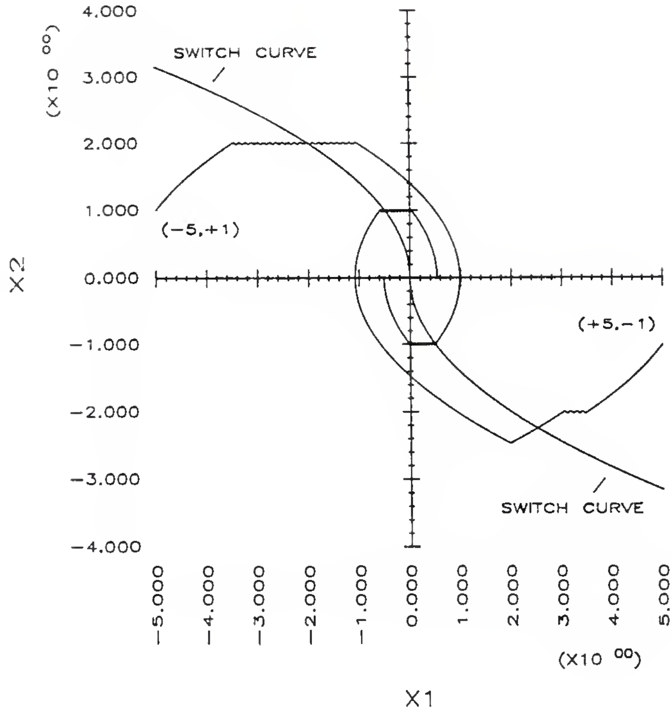


Figure 4.4.b: History plot for square grid control, set 2

The isochrone plots for grids 1, 2, 4, and 5 are all smooth, well shaped plots that are close to the true solution therefore should produce a reasonable control (see Table 4.1 for the figure number of the corresponding isochrone plots). Grid 3 has an acceptable curve shape in the doughnut region between the edges and the center. The isochrones on the edge and in the center regions are not very smooth. Grid 6 is starting to produce isochrones that are not convex, which will produce the wrong control. The number of elements is getting small enough that the approximation region for each isochrone is getting to large to provide good control. Grid 7 is given with only the element plot and isochrone plot. The control produced by this grid is oscillatory about the origin. The grid spacing has become too large to produce a stable control.

The test points for each grid give some indication on how well each grid performs. On several of the paths, the control switches early, continues for a period of time, switches again, crosses the actual switching curve, and then switches again. This produces some interesting results. After the final switch the path is very close to the true switching curve. This gives the system a better overall performance because the stopping circle will be reached directly and another cycle of switches to reach the stopping circle will not be needed. The performance of a controller decays substantially if it switches too late and has to switch again to drive the system to the stopping position. Grid 3 switches its control early, but drives the system to the final destination by chattering. Grid 6 produces

acceptable control, but the switch after crossing the switching curve comes too late, thus causing another switch to be required to drive the system to the final destination.

The grids were used to control the 500 different starting positions generated from the pseudo-random number generator. Rather than save the systems entire path history, the simulation final time, the optimal final time, and a ratio of simulated/optimal of each position was saved (the time values are the calculated time the path crossed the stopping circle of radius = 0.06). A controller based upon the switching curves was implemented for comparison purposes. This controller looks at the state position at each sample time. When the position crosses the switching curve, the control is switched. A summary of the results is presented in Table 4.2.

Using the optimal/actual ratio, the results of the simulations show the following performance order

1. Grid 4
   Grid 2
   Grid 5
   Grid 1
2. Switch Curve
3. Grid 6
4. Grid 3.

The unexpected result is the placement of the controller based upon the switching curve. This can probably be attributed to the fact that the controller based upon the switching curves always switches after crossing the switching curve. If the switch is late enough, the system will have to switch again to drive the system to the stopping circle. The controller based upon the systems isochrones often switches early

enough that the system does not need to switch again to drive the system to the stopping circle. Grids 4, 2, 5, and 1 have basically the same performance results. The results are the same for two significant digits. Grids 6, and 3 were expected to produce poorer results because of the grid density and shape.

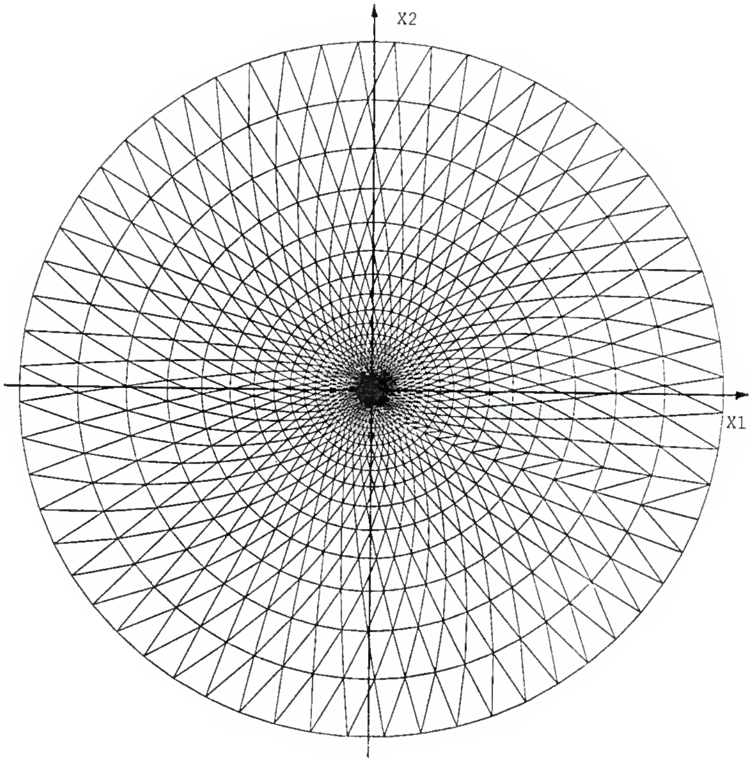
The grids produce a very good control, on the average only 2% over optimal. Smith [25] used linear segments to approximate the switching surfaces of a triple integrator. He obtained 50% to 100% over optimal. The problem he solved was more difficult and the grid he used was very coarse, so only some analogy can be reached from comparing the two.

GRID #	# SPKS	# CIRCLES	SCALE	GRIDF	ELEMENT PLOT	ISOCHRONE PLOT	HISTORY PLOT
1	60	28	28.5	1.2	Fig. 4.5.a	Fig. 4.5.b	Fig. 4.5.c & d
2	60	28	28.5	1.5	Fig. 4.6.a	Fig. 4.6.b	Fig. 4.6.c & d
3	60	28	28.5	1.0	Fig. 4.7.a	Fig. 4.7.b	Fig. 4.7.c & d
4	40	18	28.5	1.5	Fig. 4.8.a	Fig. 4.8.b	Fig. 4.8.c & d
5	30	12	28.5	1.5	Fig. 4.9.a	Fig. 4.9.b	Fig. 4.9.c & d
6	20	9	28.5	1.8	Fig. 4.10.a	Fig. 4.10.b	Fig. 4.10.c & d
7	16	9	28.5	1.8	Fig. 4.11.a	Fig. 4.11.b	n.a.

Table 4.1: Grid characteristics

	AVERAGE TF OPTIMAL		AVERAGE TF ACTUAL		OPTIMAL/ACTUAL	
	0.02	0.03	0.02	0.03	0.02	0.03
SWITCH CURVE	7.5914	7.5914	7.8424	7.9908	0.9672	0.9457
GRID 1	7.5914	7.5914	7.7462	7.9505	0.9807	0.9542
GRID 2	7.5914	7.5914	7.6799	7.8204	0.9887	0.9700
GRID 3	7.5914	7.5914	11.7522	11.7186	0.6159	0.6190
GRID 4	7.5914	7.5914	7.6787	7.8170	0.9890	0.9705
GRID 5	7.5914	7.5914	7.6780	7.8169	0.9869	0.9704
GRID 6	7.5914	7.5914	8.1558	8.1481	0.9115	0.9132

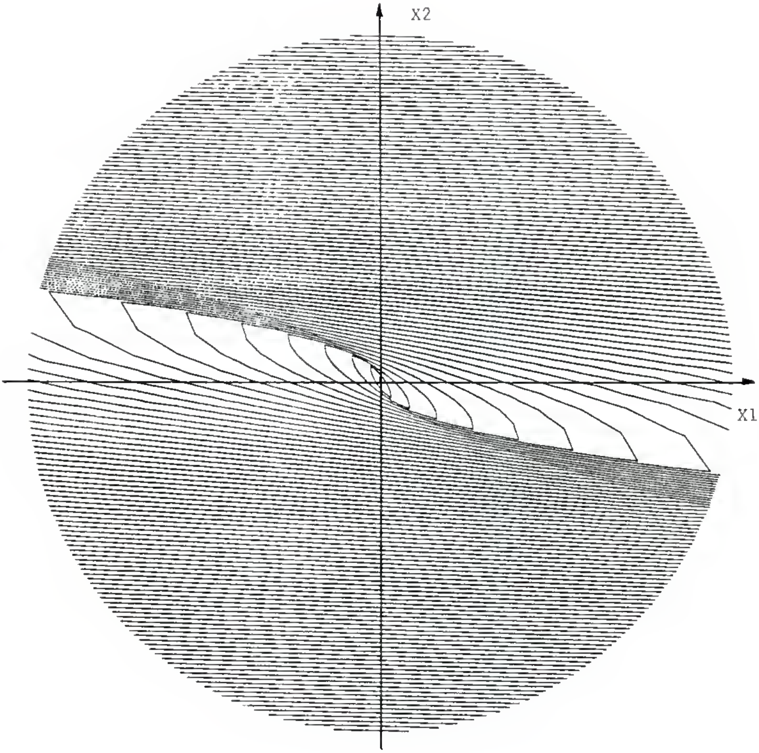
Table 4.2: Controller performance



Grid 1

Spokes	=	60
Circles	=	28
Scale	=	28.5
Grid Factor	=	1.2

Figure 4.5.a: Element plot for grid 1



Grid 1

Spokes	-	60
Circles	-	28
Scale	-	28.5
Grid Factor	-	1.2

Figure 4.5.b: Isochrone plot for grid 1

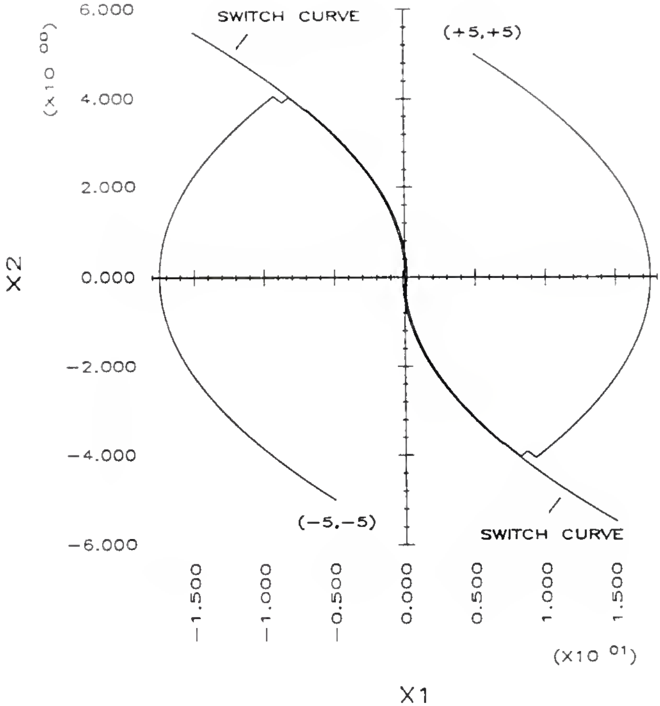


Figure 4.5.c: History plot for grid 1 control, set 1

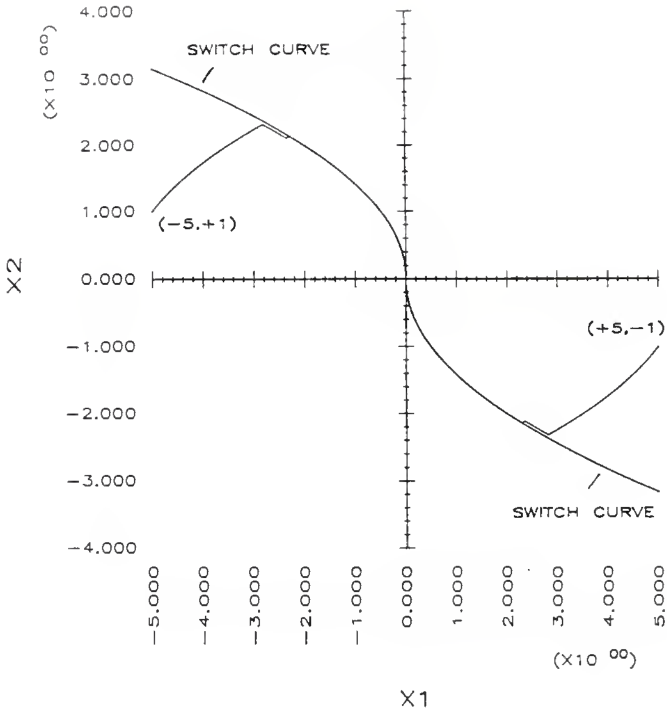
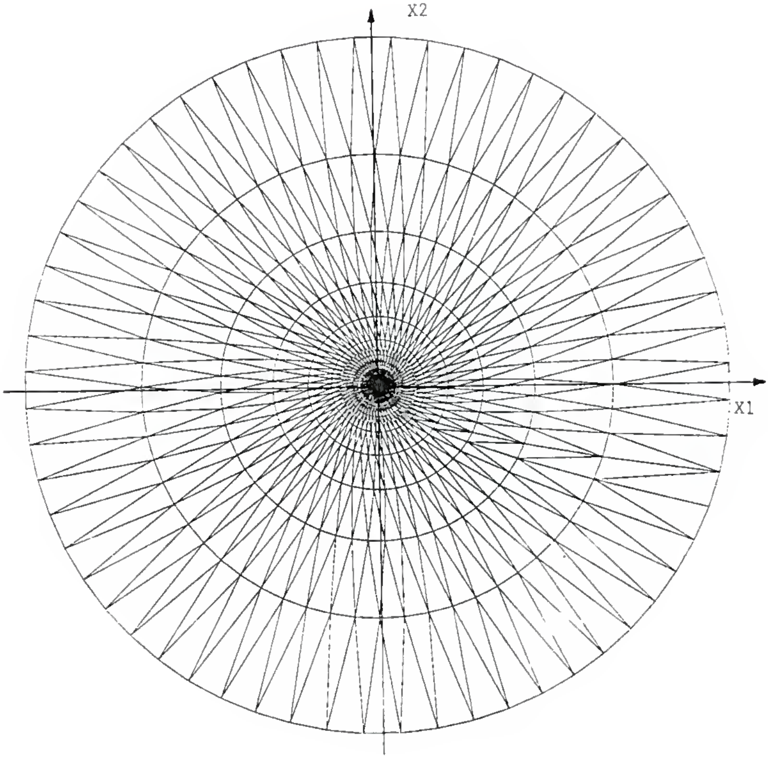


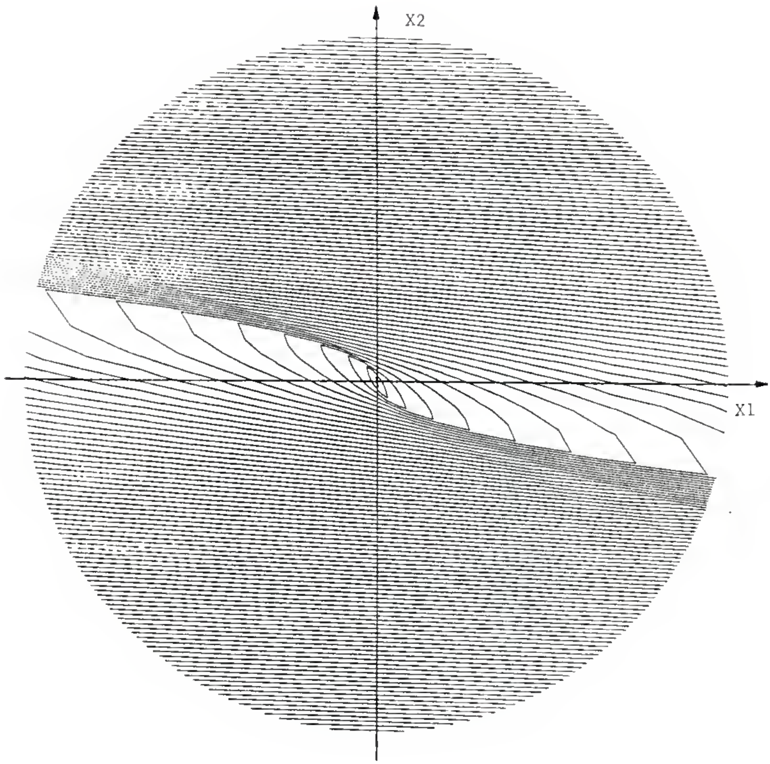
Figure 4.5.d: History plot for grid 1 control, set 2



Grid 2

Spokes	=	60
Circles	=	28
Scale	=	28.5
Grid Factor	=	1.5

Figure 4.6.a: Element plot for grid 2



Grid 2

Spokes	-	60
Gircles	-	28
Scale	-	28.5
Grid Factor	-	1.5

Figure 4.6.b: Isochrone plot for grid 2

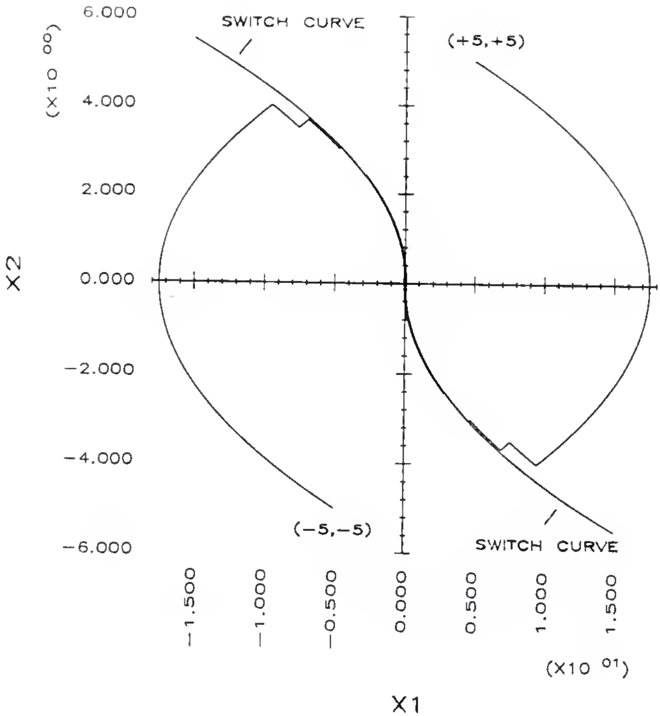


Figure 4.6.c: History plot for grid 2 control, set 1

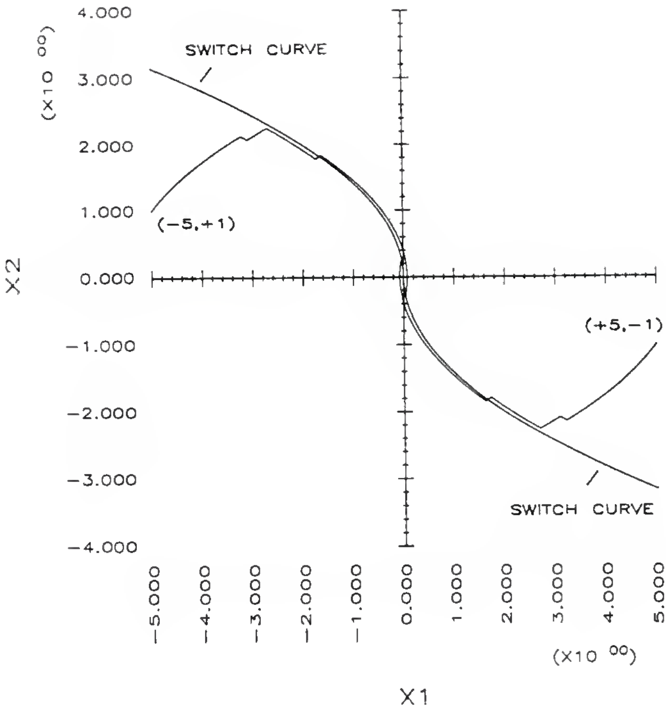
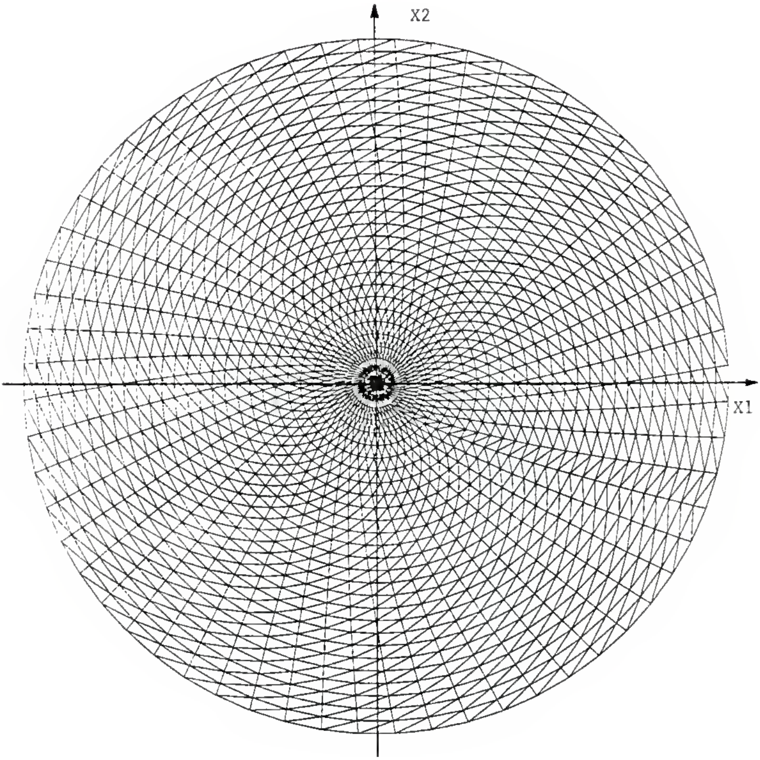


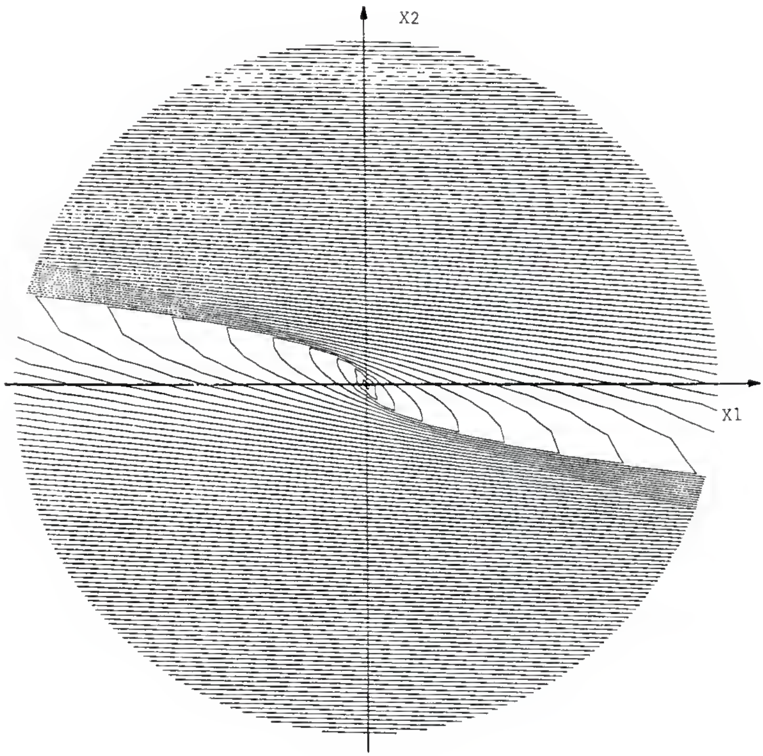
Figure 4.6.d: History plot for grid 2 control, set 2



Grid 3

Spokes	=	60
Circles	=	28
Scale	=	28.5
Grid Factor	=	1.0

Figure 4.7.a: Element plot for grid 3



Grid 3

Spokes	-	60
Circles	=	28
Scale	-	28.5
Grid Factor	-	1.0

Figure 4.7.b: Isochrone plot for grid 3

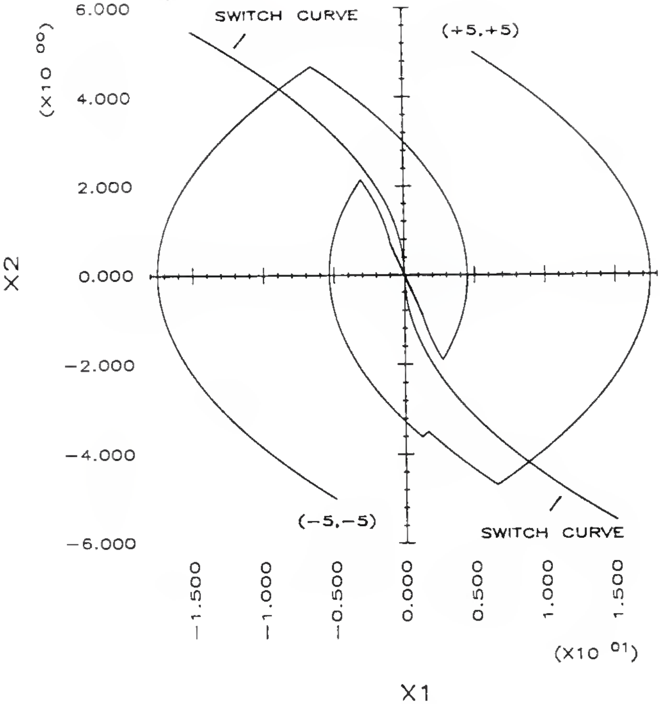


Figure 4.7.c: History plot for grid 3 control, set 1

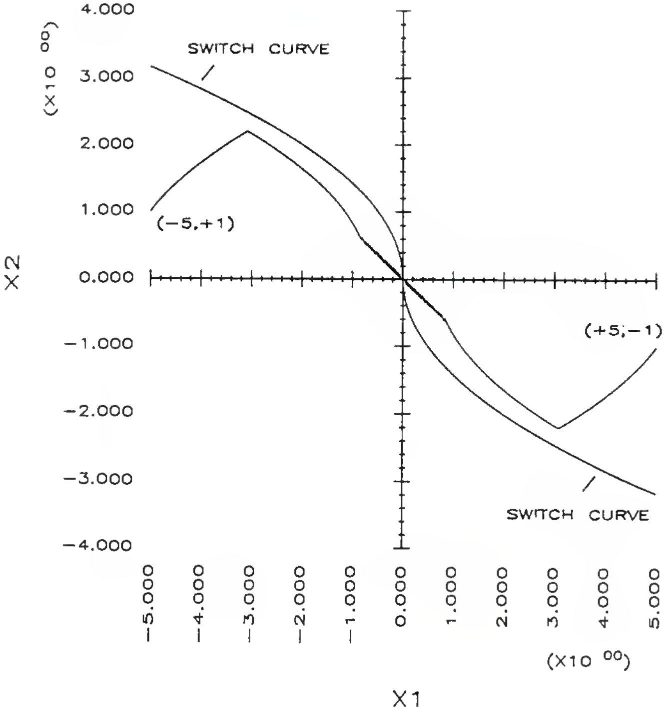
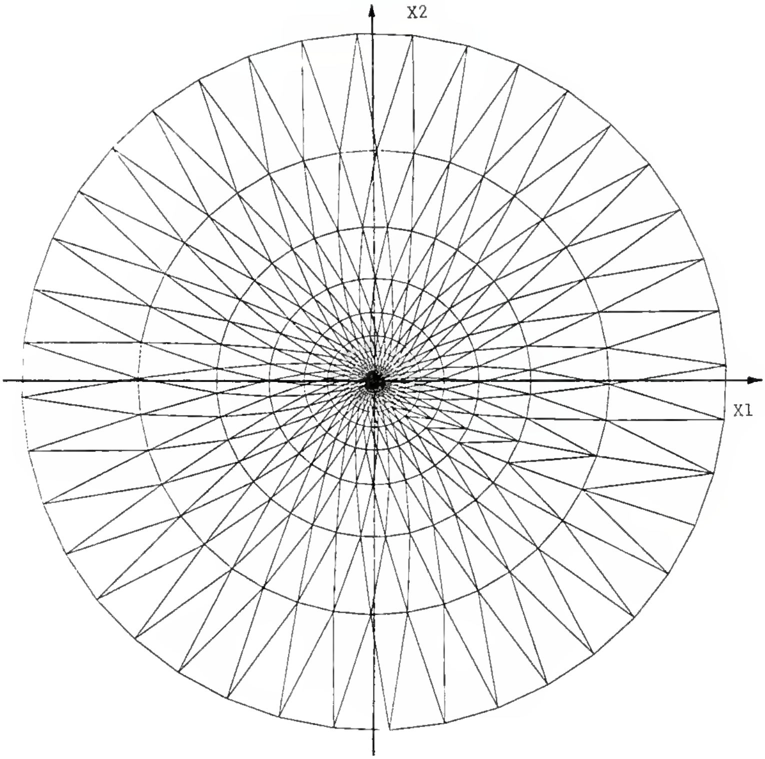


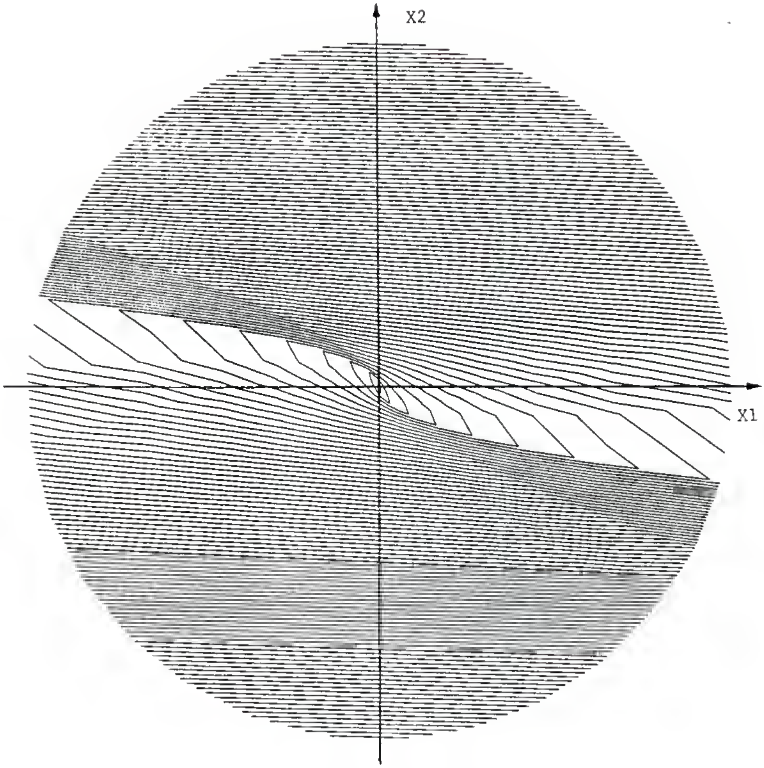
Figure 4.7.d: History plot for grid 3 control, set 2



Grid 4

Spokes	=	40
Circles	=	18
Scale	=	28.5
Grid Factor	=	1.5

Figure 4.8.a: Element plot for grid 4



Grid 4

Spokes	=	40
Circles	=	18
Scale	=	28.5
Grid Factor	=	1.5

Figure 4.8.b: Isochrone plot for grid 4

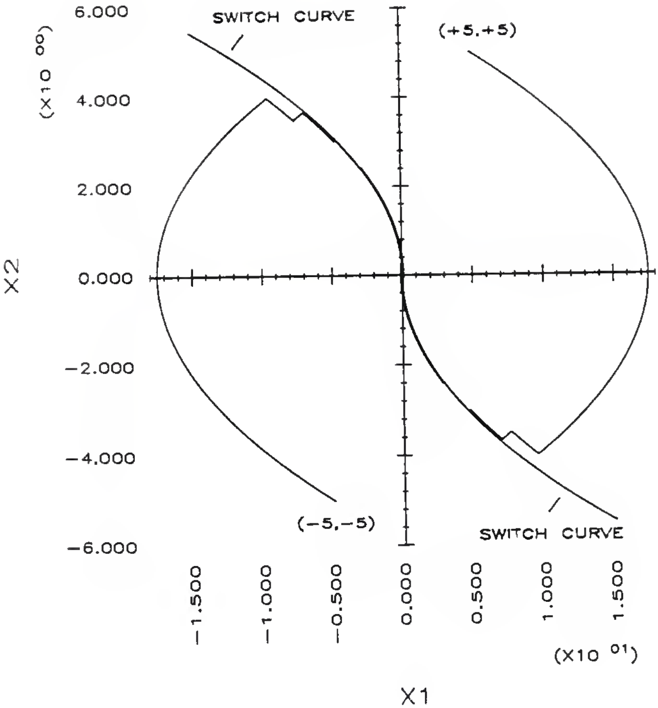


Figure 4.8.c: History plot for grid 4 control, set 1

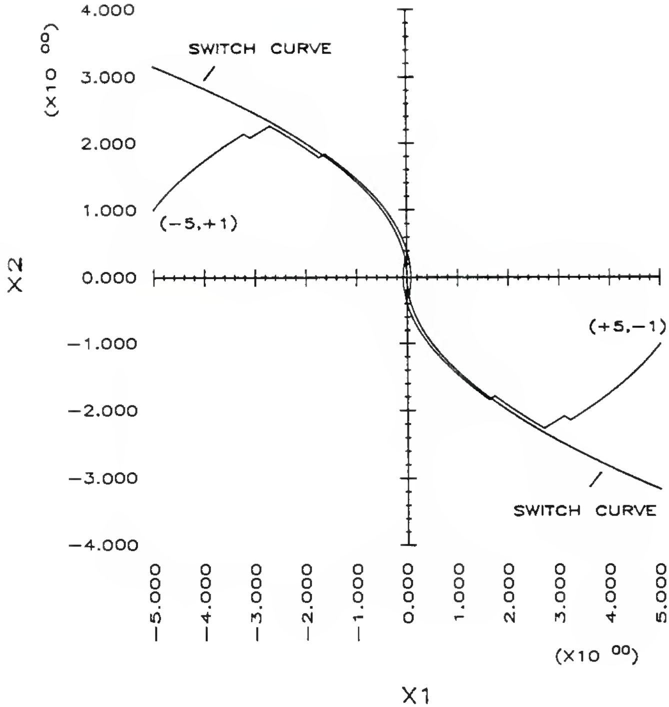
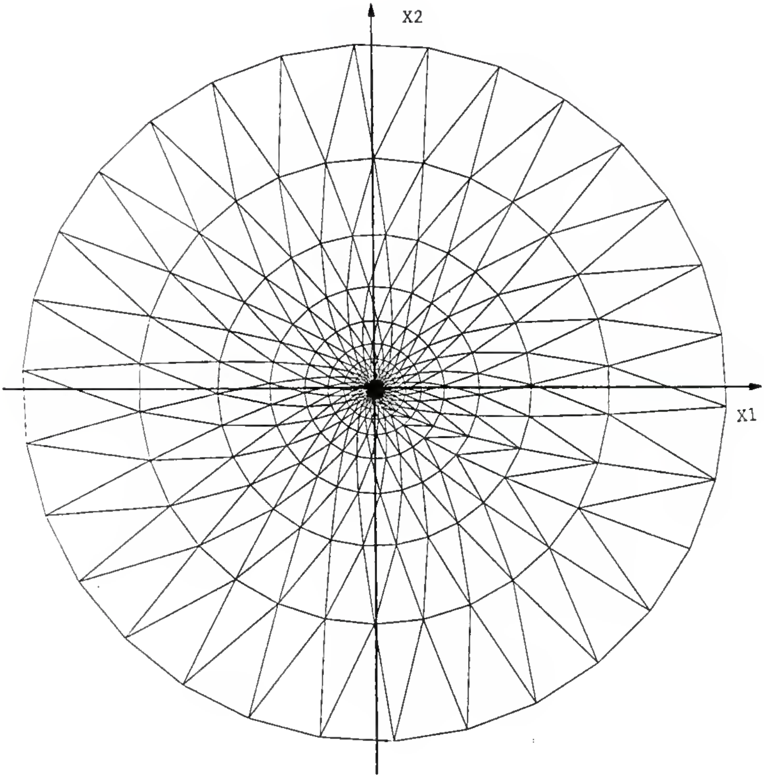


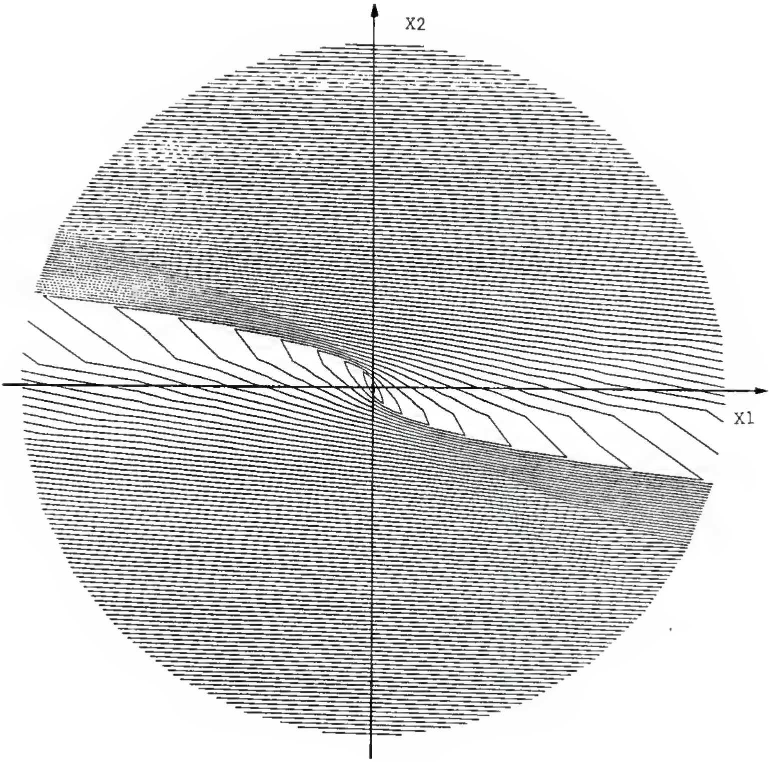
Figure 4.8.d: History plot for grid 4 control, set 2



Grid 5

Spokes	=	30
Circles	=	12
Scale	=	28.5
Grid Factor	=	1.5

Figure 4.9.a: Element plot for grid 5



Grid 5

Spokes	=	30
Circles	=	12
Scale	=	28.5
Grid Factor	=	1.5

Figure 4.9.b: Isochrone plot for grid 5

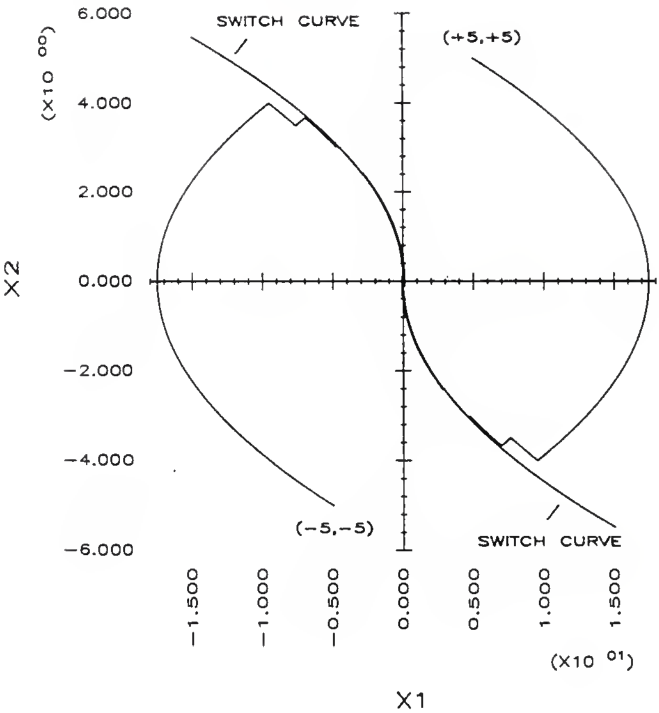


Figure 4.9.c: History plot for grid 5 control, set 1

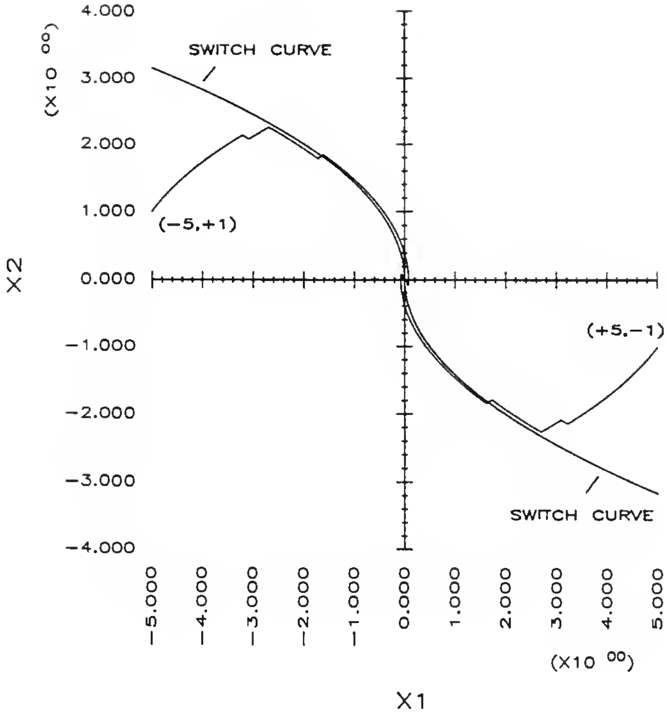
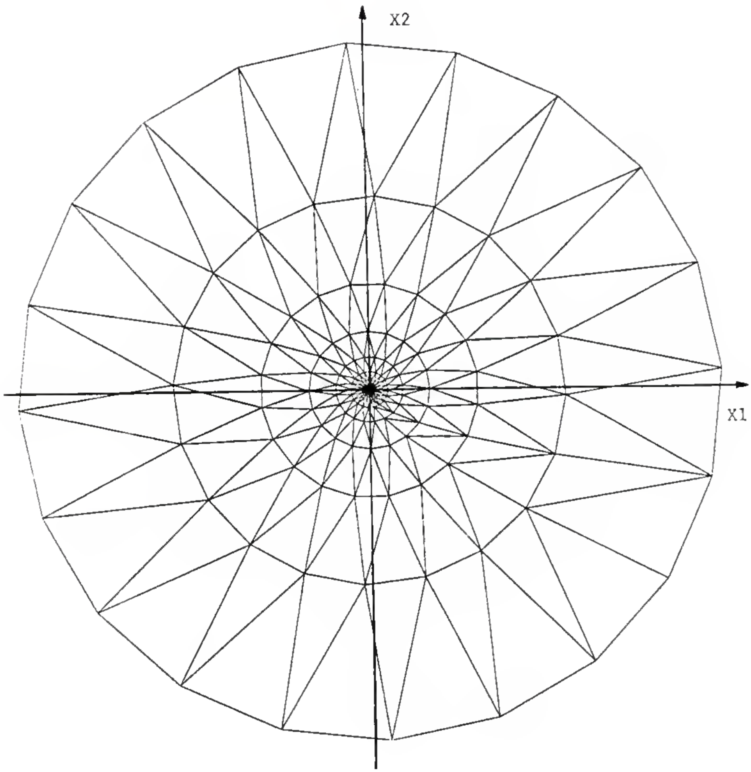


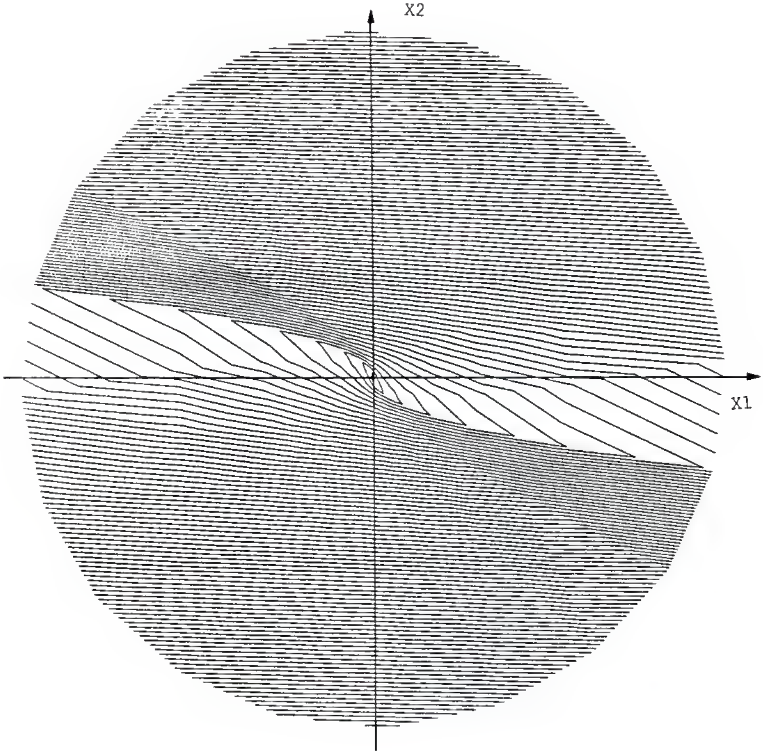
Figure 4.9.d: History plot for grid 5 control, set 2



Grid 6

Spokes	=	20
Circles	=	9
Scale	=	28.5
Grid Factor	=	1.8

Figure 4.10.a: Element plot for grid 6



Grid 6

Spokes	=	20
Circles	=	9
Scale	=	28.5
Grid Factor	=	1.8

Figure 4.10.b: Isochrone plot for grid 6

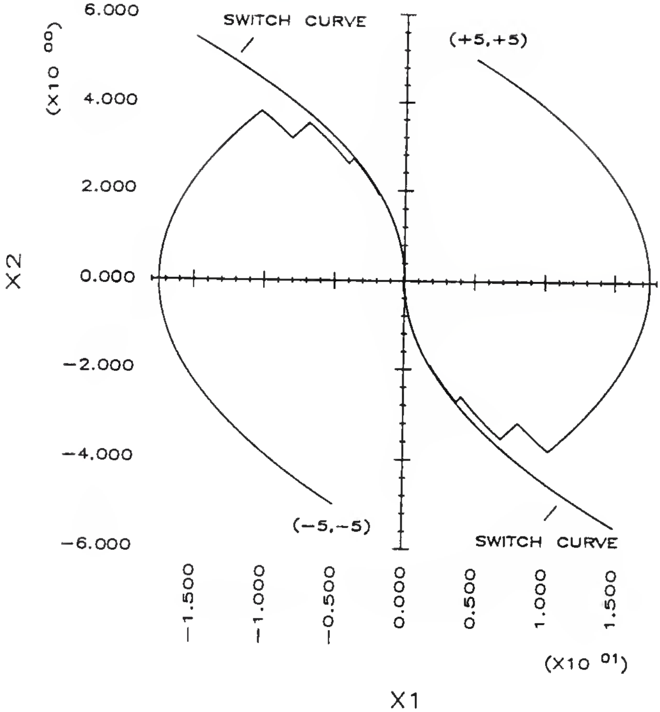


Figure 4.10.c: History plot for grid 6 control, set 1

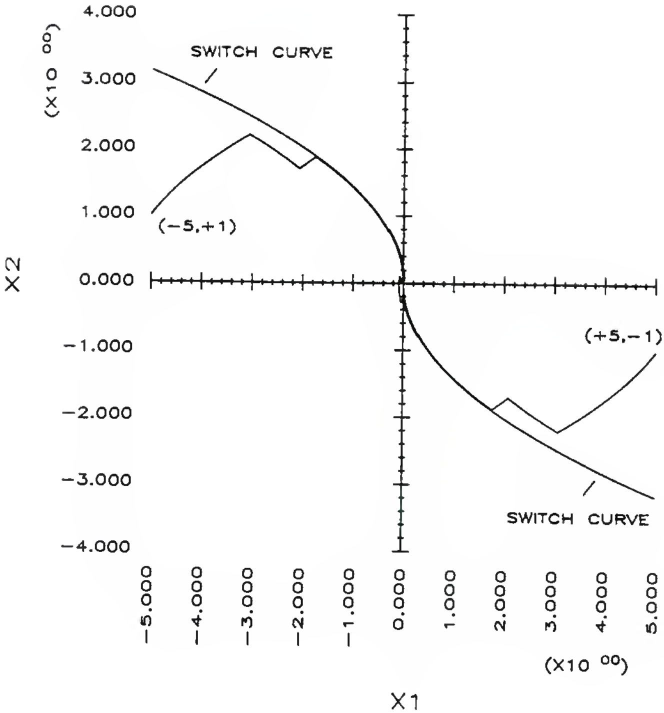
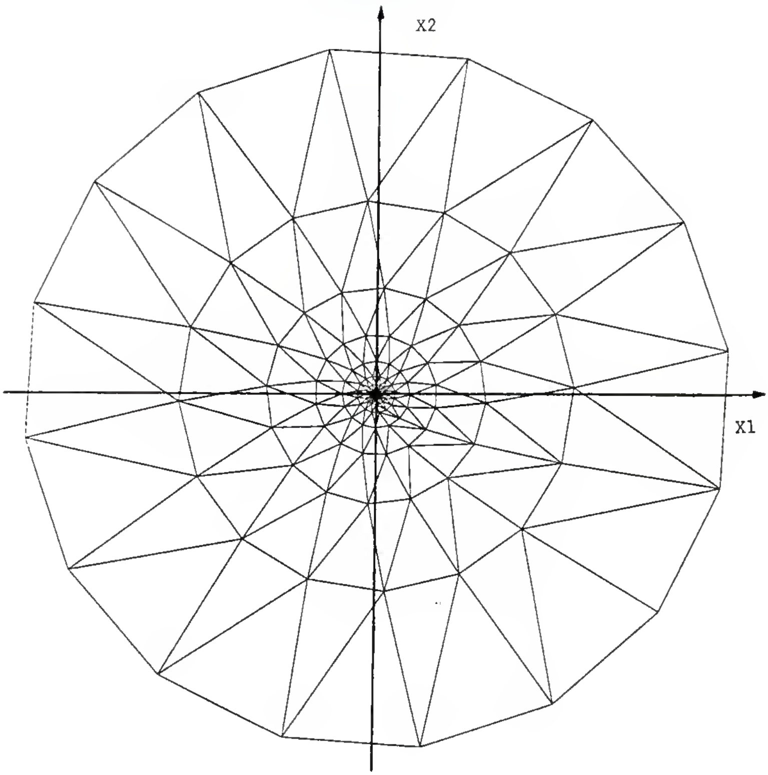


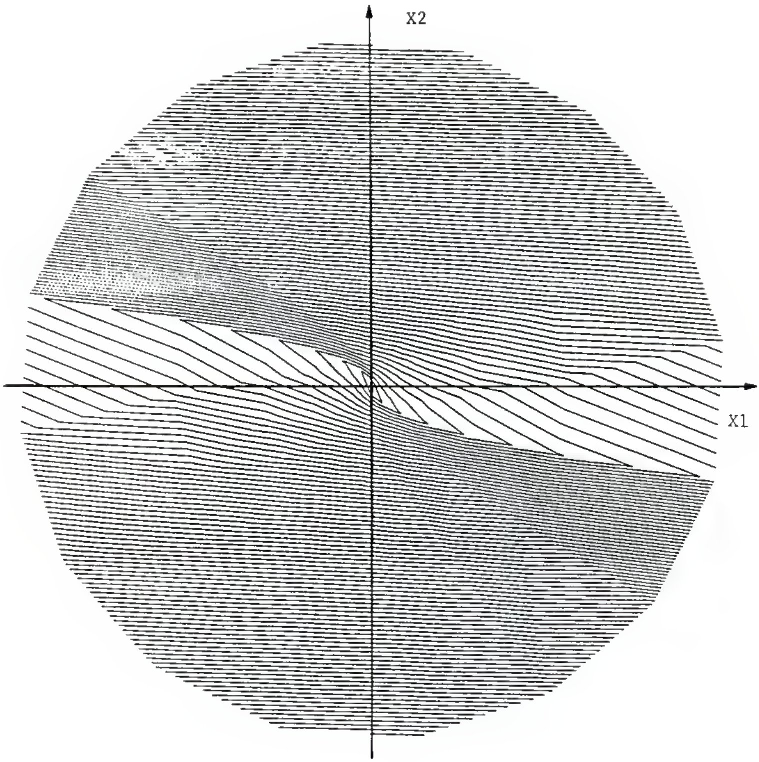
Figure 4.10.d: History plot for grid 6 control, set 2



Grid 7

Spokes	=	16
Gircles	=	9
Scale	=	28.5
Grid Factor	=	1.8

Figure 4.11.a: Element plot for grid 7



Grid 7

Spokes	=	16
Circles	=	9
Scale	=	28.5
Grid Factor	=	1.8

Figure 4.11.b: Isochrone plot for grid 7

Chapter 5

CONCLUSIONS AND RECOMMENDATIONS

Conclusions of the Investigation

The objective of this investigation has been to control a double integrator system by generating the control from the systems isochrones. The isochrones are generated from a discrete finite element final time grid. Two types of finite element grids were used, a square grid and a polar grid. Simulations were performed on two personal computers, one working as a digital controller and the other simulating a double integrator plant.

The square finite element grid was developed directly from the work of White and Rajendran [27] and Rajendran [23]. The control produced by this grid chattered in the region around the state origin. This was due to the horizontal linear segment approximation of the switching curve across the element made in that investigation.

The second grid was built using a polar grid format. The grid was built to conform to the switching curves. Using this technique, the boundary conditions along the switching curves were enforced to improve the results. The grids were generated on an Apollo DN3000 workstation. The time required to generate a dense grid solution was about one minute.

The polar grids provided a good approximation for the system's isochrones. The finite element final time grid was used to generate control for the system. The control was very good, on the average the final times achieved were only 2% over optimal. In a digital control application where time is discrete, this technique performs better than using the switching curves. When the switching curves are used, the path always crosses the curve before the control switches. If the switch is sufficiently late, another switch will be required to drive the system to the stopping point. The isochrone control system switches early and chatters once or twice until the path is closely following the switching curve. On the average this reduces the number of times the system has to switch and reverse its path to reach the stopping point.

White and Rajendran [27] had used the Hamilton-Jacobi equation and the minimum time functional together with the finite element interpolation to determine the minimum time isochrones for the system. The formulation was repeated leaving out the minimum time functional but the results produced a poor approximation for the system's isochrones.

Recommendations for Further Study

Part of the original concept of this project was to design and implement a real time controller. The finite element grid was to be solved in real time, meaning a coarse grid would be solved quickly and then used by the controller to control the system while it was resolving for the final times using a denser grid. The original

calculation rates achieved by Rajendran [23] had indicated that this control technique may have been possible but the grid developed by Rajendran did not produce an acceptable control. With the second grid, it may be possible to build a real time controller using small grid sizes. The subroutine to calculate the control requires about 0.02 seconds. If a sample rate of 0.03 seconds was used, about one third of the computer's time would be available for other work. The grids with a small number of elements provided a resonable control. Calculation times for a grid with few elements would be reduce the previous time by about a factor of ten. A computer with greater computational power as the controller would reduce both the time required to calculate the control and solve the finite element grid. There was no optimization of the code so some reduction in time may be gained by program optimization techniques.

Appendix 1

The Communication Hardware

The communication between the two computers is across a 16 bit parallel data bus. The communication is accomplished using Direct Memory Access (DMA). The communication link can transmit data at approximately 100,000 16 bit words/second.

In the following sections, a discussion of the PDMA and DMA hardware is covered first, followed by a discussion of the mechanics of the communication. The hardware to handle the communication is then covered last. The following discussion assumes some familiarity with the personal computer's internal structure, interrupt handling, and Direct Memory Access.

Both computers have a PDMA-16 Digital DMA interface card. The card is manufactured by MetraByte Corporation, 440 Myles Standish Blvd., Taunton, MA 02780, (617) 880-3000. The PDMA card simplifies the interface with the computer's DMA process.

The PDMA and DMA Hardware

The PDMA card provides external access to the following lines which are: 16 data lines, upper/lower byte data transfer direction lines, transfer request line, transfer acknowledge line, interrupt request line, 3 auxiliary digital output lines, digital voltage supply

line, and digital ground. The 16 data lines are bidirectional lines for input/output. For a complete discussion see the PDMA-16 technical reference manual [29].

The PG's DMA controller is the INTEL 8237. For a technical description of the 8237 chip see the INTEL product specifcation document [35]. The 8237 chip has a set of registers for configuring and controlling the DMA process. Only the applicable modes of each register will be discussed. The registers of user importance on the 8237 are:

1) COMMAND REGISTER - controls the overall operation of the 8237.

   I/O port address 08 hex.

2) MODE REGISTER - controls the characteristics of each of the DMA channels.

   I/O port address 0B hex.

3) MASK REGISTER - enable/disable selected DMA channels.

   I/O port address 0A hex.

4) ADDRESS REGISTER - address of the data block. Lower 16 bits of the 20 bit address.

   I/O port address (2 x Level #)

5) BYTE COUNT REGISTER - number of bytes of data to be transferred less one.

   I/O port address (2 x Level #) + 1

6) DMA PAGE REGISTER - used for upper 4 bits of the 20 bit address.

   I/O port addresses for each mode's page register is:

mode 0 and 1 at 83 hex

mode 2 at 81 hex

mode 3 at 82 hex.

There are four levels of DMA which signify the four different DMA processes available. Level 0 is used by RAM refresh, level 2 and 3 by the disk drives, and level 1 is free for use.

The PDMA board is used to transfer data by DMA from memory to its latched ports or from its latched ports to memory. The DMA process is started by a rising edge on the transfer request line, XREQ. The 8237 DMA controller requests control of the bus from the central processor. When the central processor completes its current command, it passes control to the 8237 which then begins the DMA operation. Each time the cycle is started, a falling edge pulse is sent on the transfer acknowledge line, XACK. The DMA cycle is signaled as complete on the rising edge of the transfer acknowledge line. At this point several things can happen depending on which mode the 8237 is selected. The single transfer mode is used in this work. This mode returns control to the central processor after the transfer, therefore, for each transfer request one byte is transferred. The PDMA modifies this so that if the PDMA is in word transfer mode, the DMA process will transfer 2 bytes of data for a single transfer request. Transfer acknowledge will likewise send out a single pulse to indicate a word transfer.

The PDMA's registers are:

1) PORT A - least significant byte of the data word.

2) PORT B - most significant byte of the data word.

3) DMA GONTROL REGISTER - DMA level select, enable/disable, transfer request source, byte/word transfers, direction of port A and port B, and bits for the output lines Aux 1 and Aux 2.

4) INTERRUPT CONTROL REGISTER - enable/disable interrupt, hardware interrupt level, source of the interrupt - external/internal, interrupt on the rising edge or falling edge of external input signal, and bit for the output line Aux 3.

5) COUNTERS 0, 1, and 2 - counter registers for the 8254 interval timer.

6) 8254 GONTROL REGISTER - mode control for each of the counters.

The DMA control register configures several parameters. It enables/disables the PDMA transfer request signal for the selected DMA level, selects direction of transfer, and selects the transfer request source. The transfer request sources are:

1) the external XREQ line

2) the output of the 8254 timer.

The interrupt register disable/enables the selected hardware interrupt (levels 2-7) and selects the interrupt request source. The interrupt request can come from 3 sources:

1) external input on the "int" line, +edge/-edge.

2) terminal count of the 8237.

3) output of the 8254 interval timer.

The external "int" line can be configured to generate an interrupt request on either the rising or falling edge of the input pulse. The

terminal count of the 8237 generates an interrupt when the number of bytes of data has been transferred. The 8254 timer can be set to generate a specified frequency to drive a periodic interrupt.

The Aux 1, 2, and 3 lines are three user digital output lines. They are used in the handshaking and control process.

The Mechanics of the Communication

A DMA transfer is initiated by a rising edge transition on the XREQ line. With the start of the DMA cycle, the transfer acknowledge line, XACK goes low. At the completion of the DMA cycle, the XACK line returns high. The basis of the communication cycle is to use the XACK of the sending board to drive the XREQ of the receiving board. The XREQ frequency of the sending board is set to a slow enough rate so that the receiving board has completed its cycle before the sending board sends another word. This process can then be repeated to send unidirectional data.

To reverse the direction of the transfer, one technique would be to call the various hardware initialization subroutines to change the appropriate registers. To change the 8237 registers and the PDMA registers using the DMA setup subroutine takes about 0.2 seconds. This is to slow for a real time simulation. The 8237 has a recycle mode that reloads the address and byte count registers after the byte count. register reaches 0, the terminal count. In this mode, changing direction only requires toggling the direction bits on the 8237 and PDMA board, just 2 registers to change. The transfer request signal would then be used to drive the second computer and its transfer

acknowledge would drive the transfer request of the first computer. Handshake hardware between the two computers would handle this change.

The process of changing the two registers to change the direction is a very fast technique. Using this process requires that all the other registers remain the same. The address and byte count registers must also remain the same. The actual data transferred between the two computers is a communication vector. This vector is made up of the current state and control, three floating point numbers. The entire communication vector is transmitted each time a transfer cycle takes place. Even though more data is being communicated than need be, it is still faster than conventional techniques.

The handshake hardware has several duties;

1) The data ports of the two PDMA boards are separated by tristate lines. The direction of the transfer is set according to the direction outputs of each board. During the direction change process it is possible for both ports to be in a output mode. If this condition occurs, the ports must be separated to prevent damage to either board.

2) When both computers are ready, the hardware starts the DMA cycle. The Aux 1 line is used to signal a ready state from the computer. The DMA cycle is controlled by the input clock frequency to the sending board's XREQ. The frequency is generated by the PDMA's counter chip outputs. The frequency should be slow enough so that the receiving computer can finish transferring the data before the start of the next cycle. The transfer request frequency input to the sending

board should be n periods long where n is the number of data transfers. After the nth transfer, the 8237 reloads all its registers to their initial values as defined by the recycle mode. After the nth request the 8237 reloads all its registers and is then ready to send the same set of data.

3) Control the handshaking between the two PDMA boards by connecting the transfer request output to the sending PDMA's transfer request and connecting the transfer acknowledge of the sending PDMA to the receiving PDMA's transfer request.

4) Send the sample rate interrupt to the "control system", the Zenith.

The handshake hardware has to be a very "clean" system. Any noise on the transfer request or transfer acknowledge lines will cause spurious DMA initiations, thus putting the 2 computers out of phase. The PDMA's transfer request line was found to be very sensitive to noise thus requiring the hardware to be carefully designed.

The Handshaking Hardware

The following discussion will be broken down into four major sections. The four sections correspond to the four requirements of the handshake hardware discussed in the previous section.

The first section of the hardware shows the 40 pin interface between the two computers (see Figure A1.2). It also contains the data bus connection between the two computers. The tristate buffering is provided by an octal bus transceiver, the 74HC245 chip. The chip has connections for 8 data lines. Two chips are used to buffer the 16 bit

word; the most significant byte consists of lines D15-D8 and the least significant byte consists of lines D7-D0. The data direction is controlled by a DIR input and the enable is controlled by a $\bar{G}$ input. The logic diagram is shown in Table A1.1.

The direction pin is required to be at the correct state at the transfer time only. The enable is active only when the directions of the two boards are opposite, this is to prevent damage to either board.

Zenith DIR	HP DIR	Chip DIR	G
0	0	X	1
0	1	0	0
1	0	1	0
1	1	X	1

Table A1.1: Logic diagram of data bus enable

The outputs of the two boards used by the handshake hardware are buffered through a 74HC245 chip to provide protection and a higher current output.

The output connection of the PDMA board is a 37 pin D type male connector. A 40 wire bus was used and connected with a 40 pin low profile male dip connector. This was used to provide an easy interface with the handshake hardware board. The pin numbering is different due to this translation. The pin numbering is shown in Table A1.2.

The second section of the hardware generates the transfer request pulse train, XFREQ (see Figure A1.3). This pulse train is generated when both computers are ready to transmit/receive. The computers set

the Aux 1 line to an on state to signal ready, ZAUX1 for the Zenith, and HPAUX1 for the Hewlett Packard. A D-type positive edge triggered flip-flop (74LS74) is used to sense when both signals are at a high state. The clear is used to send a logic one when both AUX 1 lines have gone low and back high again. This is used because the Hewlett Packard's cycle time was considerably faster than the Zenith's. When the outputs of the Aux 1 lines have cycled high, a monostable multivibrator (74121) generates a pulse. This pulse is used to load a set of cascaded up/down counters (74LS193) with a preset number. This number is determined by a set of DIP switches. The counters then count down to zero and stop. The clock on the counters is driven by the output of the Zenith's PDMA 8254 interval timer, ZTOUT. The output of the least significant bit of the counters is the pulse train with the desired number of pulses, XFREQ. Since the least significant bit is used to generate the pulse train, there will be (number/2) pulses generated at a frequency of (TOUT/2) Hz. Another up/down counter is used to count the output pulses of the least significant bit. The output of this counter is displayed using LED's. This enables the user to check the settings of the DIP switches during operation.

PIN # ON 37	PIN # ON 40	FUNCTION
19	22	A0
37	18	A DIR. OUT
18	23	A1
36	17	GND
17	24	A2
35	16	GND
16	25	A3
34	15	GND
15	26	A4
33	14	GND
14	27	A5
32	15	GND
13	28	A6
31	12	GND
12	29	A7
30	11	GND/INT 4
11	30	B0
29	10	B DIR. OUT
10	31	B1
28	9	GND
9	32	B2
27	8	GND
8	33	B3
26	7	AUX 3 OUT
7	34	B4
25	6	AUX 2 OUT
6	35	B5
24	5	AUX 1 OUT
5	36	B6
23	4	TIMER GATE IN
4	37	B7
22	3	TIMER OUT
3	38	XFER. ACK. OUT
21	2	+5v
2	39	XFER. REQ. IN
20	1	+5v
1	40	INTERRUPT IN

Table A1.2: Pin out for the 37 and the 40 pin connectors

The third section of hardware controls the inputs to the transfer request for the Zenith, ZXREQ and for the Hewlett Packard, HPXREQ (see Figure A1.4). The process is to send the XFREQ signal to the sending computer's XREQ and to transmit the sending computer's XACK to the receiving computer's XREQ. This sets up the handshaking process. The XACK of each computer is first sent to a monostable multivibrator (74121). The natural state of the XACK is high. It was discovered that if XREQ is high when the DMA is initialized, it senses a rising edge since the natural state of XREQ is low. By using the monostable, a <u>XACK</u> is set up with a natural state of low. A logic network is used to set up the the proper XREQ inputs. (NOTE: <u>XACK</u> = the output of the monostable multivibrators). The Zenith's transfer request, ZXREQ, is determined by the truth table of Table A1.3. The Hewlett Packard's transfer request, HPXREQ, is determined by the truth table of Table A1.4.

The direction bit for the Hewlett Packard, HPDIR and for the Zenith, ZDIR corresponds to 1 as the output state, and 0 as the input state.

ZDIR	<u>HPXACK</u>	XFREQ	ZXREQ
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table A1.3: Zenith transfer request logic table

HPDIR	<u>ZXACK</u>	XFREQ	HPXREQ
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table A1.4: Hewlett Packard tranfer request logic table

The fourth section of the hardware is basically a pulse stretcher (see Figure A1.5). The output of the Hewlett Packard's PDMA 8254 interval timer, HPTOUT, is used to generate the sample frequency. The frequency generates a hardware interrupt at each sample time. The pulse width of the timer's output is on the order of 10 nanoseconds. The interrupt controller (INTEL 8259A) on the PC requires a minimum pulse width of 350 nanoseconds. A monostable multivibrator (74121) is used to stretch the timer's pulse width to about 300 microseconds. The pulse is tristated so that the output of the circuit only affects the PC's backplane during the pulse output time. This circuit is a stand alone circuit. It could be implemented on its own expansion card. Rather than using an expansion slot on the PC, the circuit was built onto the Zenith's PDMA board. The ground line on pin 30 of the 37 pin output was modified to be the input for this circuit.

The first three hardware sections were assembled on an external board. The board layout is shown in Figure A1.6. The board has a front and back-plane of +5V and ground respectively to help keep the noise to a minimum. Wire wrapping was used to construct the circuit.

Capacitors were used throughout the circuit to minimize voltage and ground spikes.

PARTS LIST

Integrated Circuit Components

DEVICE	PART #	FUNCTION	MFG.	#PINS
U1	74LS193	UP/DOWN COUNTER	SIGNETICS	16
U2	74LS193	UP/DOWN COUNTER	SIGNETICS	16
U3	74LS193	UP/DOWN COUNTER	SIGNETICS	16
U4	74LS74	D-TYPE F/F	NAT. SEMI.	14
U5	74121	MONOSTABLE MULTIV.	NAT. SEMI.	14
U6	74LS32	2-INPUT OR	NAT. SEMI.	14
U7	74LS08	2-INPUT AND	NAT. SEMI.	14
U8	74LS00	2-INPUT NAND	NAT. SEMI.	14
U9	74121	MONOSTABLE MULTIV.	NAT. SEMI.	14
U10	74121	MONOSTABLE MULTIV.	NAT. SEMI.	14
U11	74LS00	2-INPUT NAND	NAT. SEMI.	14
U12	74LS08	2-INPUT AND	NAT. SEMI.	14
U13	74HC245	OCTAL BUS TRANS.	NAT. SEMI.	20
U14	74HC245	OCTAL BUS TRANS.	NAT. SEMI.	20
U15	74HC245	OCTAL BUS TRANS.	NAT. SEMI.	20
U16	74126	TRISTATE BUFFER	NAT. SEMI.	14
U17	74121	MONOSTABLE MULTIV.	NAT. SEMI.	14

Capacitors

DEVICE	VALUE (uF)	DESCRIPTION	QTY.
C1	0.22	POLYCARBONATE	4
C2	0.1	POLYCARBONATE	10
C3	10	TANTALUM	9
C4	22	TANTALUM	2
C5	0.1	CERAMIC	12
C6	73.2 pF	MICA	1
C7	2200 pF	CERAMIC	6

Resistors

DEVICE	VALUE (ohms)	DESCRIPTION	QTY.
R1	390	5% CARBON	6
R2	2.2K	5% CARBON	1
R3	4.7K	5% CARBON	2
R4	10K	5% CARBON	2
R5	1K	5% CARBON	1

CROSS REFERENCE OF INTEGRATED CIRCUIT PIN LOCATIONS TO DRAWING PACE

DEV.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
U1	X			X	X			X	X	X			X	X	X	X				
U2	X	X	X	X	X	X	X	X	X	X	X			X	X	X				
U3	X		X	X	X			X	X	X	X		X	X	X	X				
U4	X	X	X	X	X		X		X	X	X	X	X	X						
U5			X		X	X	X			X	X			X						
U6	X	X	X	Z	Z	Z		O	O	O	O	O	O							
U7	X	X	X	X	X	X		Z	Z	Z	Z	Z	Z							
U8	X	X	X					X	X	X	X	X	X							
U9			O		O	O	O			O	O			O						
U10			O		O	O	O			O	O			O						
U11	O	O	O	O	O	O		Z	Z	Z	Z	Z	Z							
U12	O	O	O	O	O	O		O	O	O	O	O	O							
U13	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
U14	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
U15	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
U16	A	A	A																	
U17			A		A	A	A			A	A			A						

SYMBOL		DRAWING
Z	=	BUS
X	=	X_FREQ
O	=	X_REQ
A	=	INT

Drawing Symbol Notes

⏚	DIGITAL GROUND
>—	CIRCUIT INPUT
←	CIRCUIT OUTPUT
o—	ON-BOARD CONNECTION
V_s	DIGITAL SUPPLY VOLTAGE

Figure A1.1: Symbol definitions for circuit diagrams

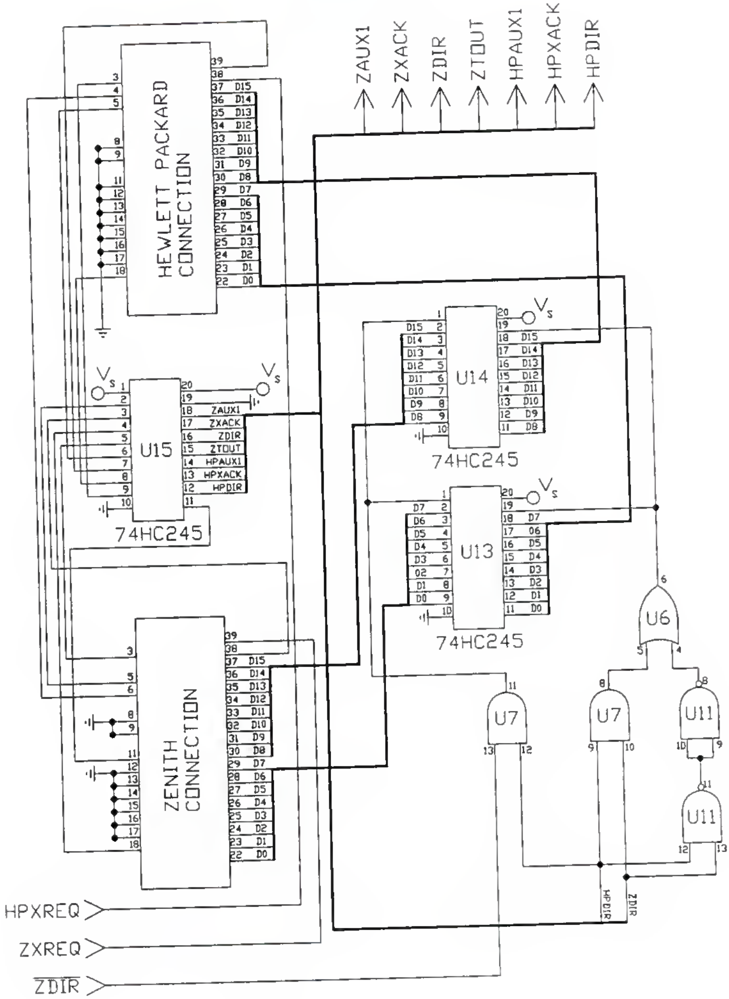


Figure A1.2: Circuit diagram of the PDMA interface

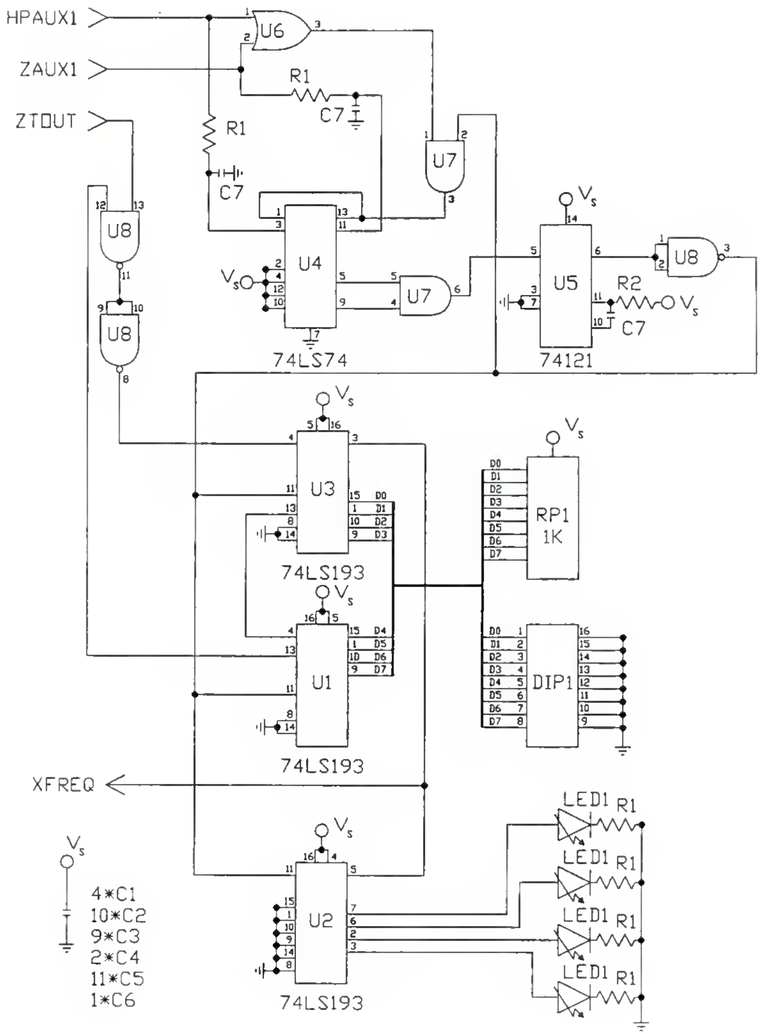


Figure A1.3: Circuit diagram of transfer request frequecy generator

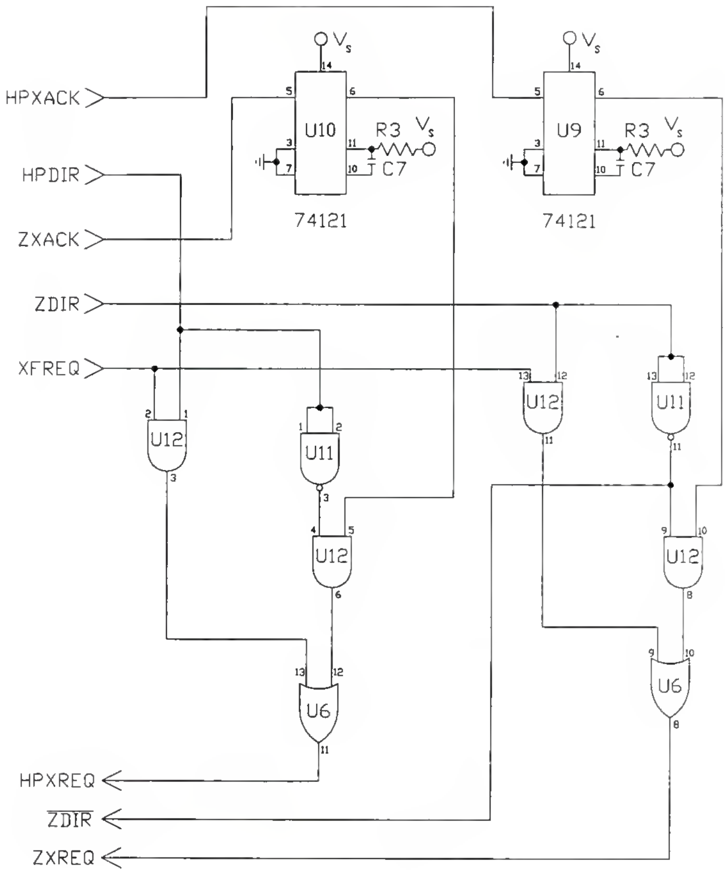


Figure A1.4: Circuit diagram of transfer request controller

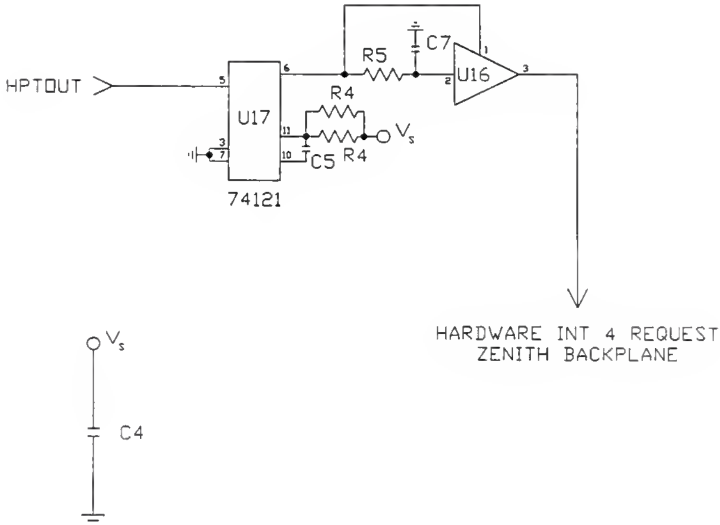


Figure A1.5: Circuit diagram of interrupt frequency filter

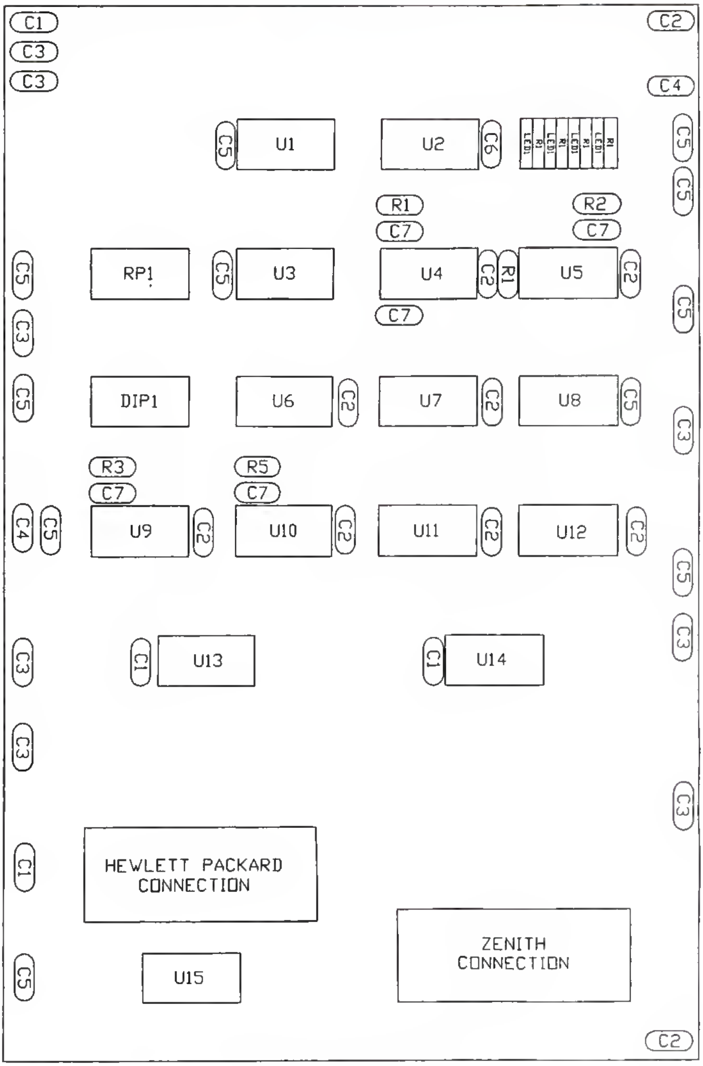


Figure A1.6: Circuit board layout

Appendix 2

The Computer Test System Software

ZSIMUL and ZRSIMUL

ZSIMUL and ZRSIMUL are the double integrator simulation programs. Both are small model programs. All the calls to the hardware set up programs are in the small model format.

```
/**********************************************************
*
*   SIMULATION HEADER FILE
*
*   information in this header is used by both HPSIMUL,
*   ZSIMUL, and ZRSIMUL.
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/22/88
*
**********************************************************/

/*      S_RATE = 1  -- sample rate 0.01
               = 2  -- sample rate 0.02
               = 3  -- sample rate 0.03     */
#define S_RATE 3

#define UMAX 1.0
#define MULT_1 1000
#define TOPTCIRCL  5.9973043E-2

#if S_RATE == 1
    #define SAMPLE     0.01
    #define MULT_2     100
#elif S_RATE == 2
    #define SAMPLE     0.02
    #define MULT_2     200
#elif S_RATE == 3
    #define SAMPLE     0.03
    #define MULT_2     300
#elif S_RATE == 4
    #define SAMPLE     2.0
    #undef  MULT_1
    #define MULT_1     1000
    #define MULT_2     20000
#endif


/* define transmission rate dividers   */
/*  best speed to date divisors - (9,6)=185.2 kHz; 10,10 good   */
#define TRANS1_DIV   10
#define TRANS2_DIV   10

/* define functions */
#define absval(a)  ( ( (a) > 0 ) ? (a) : -(a) )

void eoi_int(void);

void auxlon(void);
void auxloff(void);
```

```
void aux2on(void);
void aux2off(void);
void aux3on(void);
void aux3off();

void send(void);
void receiv(void);

void rkg( int, float, float, float *, float *, float *);
float control( float *, float);

int key_push(void);
```

```
/************************************************************
*
*   ZSIMUL -- simulation program for Zenith
*
*   zsimul sets up a simulation of a digital control
*   scheme.
*   Two interrupt routines are installed.  1) FREQ is the
*   interrupt generated by sample rate, and
*   2) TERMINAL_COUNT is the interrupt generated each time
*   the specified number of DMA transmissions has occurred.
*   The external line AUX 1 is used to signal when the
*   computer is ready to transmit or receive.  AUX 2 is
*   used to enable the sample frequency.
*
*   AUTHOR:      Donald A. Smith
*   DATE:        6/21/88
*
************************************************************/

#define LINT_ARGS
#include <utility.h>
#include <stdlib.h>
#include <bisr.h>
#include <dos.h>
#include <malloc.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include "loc.h"

float x[10];          /* x[0]=state variable x1
                         x[1]=state variable x2
                         x[2]=control u       */
int comm_bit = 0;     /* test variable for communication
                         with the interrupt routines      */

/* FUNCTION DEFINITIONS                      */
void freq(ALLREG *, ISRCTRL *, ISRMSG *);
void terminal_count(ALLREG *, ISRCTRL *, ISRMSG *);
float intersection( float, float, float, float);

main()
{
    /*      PDMA base address                */
    int base_add = 0x300;
    /*      DMA mode                         */
    int dma_level = 1;
    /*      hardware interrupt #5 for 'terminal_count   */
    int tc_level = 5;
    /*      hardware interrupt #4 for 'freq'    */
```

```
int freq_level = 4;
/*      source of 'terminal_count' interrupt    */
int tc_sourc = 2;
/*      source of 'freq' interrupt  */
int freq_sourc = 4;
/*      number of DMA transmissions */
unsigned numb_trans = 6;
/*      transmit words instead of bytes */
unsigned length = 1;
/*      initialize to output            */
unsigned direction = 1;
/*      enable DMA recycle          */
unsigned recycle = 1;
/*      transmit request external   */
unsigned trans_sourc = 0;
unsigned data_segment;

unsigned char work1,work2;
unsigned int work3;

struct SREGS segregs;

/*      pointers to history storage arrays  */
float *b0, *b1, *b2;

float t_final_theo, t_cross, switch_pt;
double p0, p1;

extern float x[];
extern int comm_bit;

int i,j;
char char_at_keybd;
FILE *out_put;
char out_file[20];
char string[80];

char_at_keybd = '!'; /* initialize to something else */

/*      disable DMA mode 1          */
outpt(0x0a,0x05);
/*      hardware check              */
if ( mode0(base_add, dma_level, tc_level))
    fprintf(stderr,"error on termination of mode0\n");

/*      set hardware communication line off */
aux1off();
/*      disable sampling frequency from the HP  */
aux2off();
```

```
fprintf(stderr,"x(0) starting?\n");
gets(string);
x[0] = atof(string);

fprintf(stderr,"x(1) starting?\n");
gets(string);
x[1] = atof(string);

fprintf(stderr,"output file?\n");
gets(out_file);
out_put = fopen( out_file,"w");

/*      calculate final time theoretical    */
switch_pt = x[0]+x[1]*absval(x[1])*0.5/UMAX;
if( switch_pt < 0.0 ) {
    p0 = -1.0/(sqrt( (double) (0.5*x[1]*x[1] -
            UMAX*x[0]) ));
    p1 = ( -1.0 - p0*x[1] )/UMAX;
    t_final_theo = 2.0*p0*x[0] + p1*x[1];

} else if( switch_pt > 0.0 ) {
    p0 = 1.0/(sqrt( (double) (0.5*x[1]*x[1] +
            UMAX*x[0]) ));
    p1 = ( 1.0 + p0*x[1] )/UMAX;
    t_final_theo = 2.0*p0*x[0] + p1*x[1];

} else {
    t_final_theo =  x[1];
}

/*      allocate space to store path and control
        history                                  */
work3 = ((t_final_theo*3.0 )/SAMPLE) * sizeof(float);
b0 = (float *) malloc(work3);
b1 = (float *) malloc(work3);
b2 = (float *) malloc(work3);
if ( b0 == NULL || b1 == NULL || b2 == NULL ) {
    fprintf(stderr,"could not allocate memory blocks\n");
    if( b0 != NULL )
        free(b0);
    if( b1 != NULL )
        free(b1);
    if( b2 != NULL )
        free(b2);
    exit();
 }

fprintf(stderr,"\nsample rate= %4.2f\n\n",SAMPLE);
fprintf(stderr,"return to continue ->\n");
gets(string);
```

```
utintoff();
/*      install interrupt routine 'terminal_count'  */
mode7( base_add, tc_level, tc_sourc, terminal_count, 1);
/*      install interrupt routine 'freq'    */
mode7( base_add, freq_level, freq_sourc, freq, 1);
/*      set output frequency for transmission rate  */
mode3( base_add, TRANS1_DIV , TRANS2_DIV);
utinton();

/*      get x() segment             */
segread( &segregs );
data_segment = segregs.ds;
/*      set up DMA                  */
model(base_add, dma_level, numb_trans, length,
      direction, recycle, trans_sourc, data_segment, x);

/*      save initial position       */
b0[0] = x[0];
b1[0] = x[1];
b2[0] = x[2] =  0.0;

send();
comm_bit = 0;

fprintf(stderr,"start\n");
fprintf(stderr,"x0=%f x1=%f x2=%f\n",x[0],x[1],x[2]);
fflush(stderr);

for( i=1; ; i++ ) {
    /*      check if exceeded history buffer    */
    if( i+10 > t_final_theo*3.0/SAMPLE ){
        fprintf(stderr,"out of room\n");
        goto kickout; /* exit the simulation    */
    }

    /*      first time start sample rate    */
    if ( i == 1 )
        aux2on();

    /*      wait for sample interval or abort   */
    while( comm_bit == 0 ) {
        if ( key_push() ) {
            char_at_keybd = getch();
            if ( char_at_keybd == 's' )
                goto kickout; /* exit the simulation */
        }
    }

    /*      turn on line ready to send vector */
    aux1on();
```

```
        /*      wait until position sent out or abort   */
        while( comm_bit == 1 ) {
            if ( key_push() ) {
                char_at_keybd = getch();
                if ( char_at_keybd == 's' )
                    goto kickout;  /* exit the simulation */
            }
        }

        /*      turn on line ready to receive vector    */
        auxlon();

        /*      wait until vector has been received or
                abort                                   */
        while( comm_bit == 2 ) {
            if ( key_push() ) {
                char_at_keybd = getch();
                if ( char_at_keybd == 's' )
                    goto kickout;  /* quit simulation   */
            }
        }

        /*      if no error save current position &
                control                               */
        if ( comm_bit == 3 ) {
            b0[i] = x[0];
            b1[i] = x[1];
            b2[i] = x[2];

        } else {
            fprintf(stderr,"error, comm_bit=%d\n",comm_bit);
            break;
        }

        /*      check for stopping criterion    */
        if( x[0]*x[0] + x[1]*x[1] < 0.06*0.06 )
            break;

        comm_bit = 0;
    }

kickout: /* exit simulation             */
    /*      turn off DMA mode 1         */
    outpt(0x0a,0x05);
    /*      turn off sampling frequency */
    aux2off();

    utintoff();
    /*      uninstall interrupts        */
    mode7( base_add, tc_level, tc_sourc, terminal_count, 0);
```

```
    mode7( base_add, freq_level, freq_sourc, freq, 0);
    utinton();

    /*      calculation of exact time crossed exit
            criterion                                   */
    if(comm_bit == 3)
        t_cross = intersection( b0[i-1], b1[i-1], b2[i-1],
                (i-1)*SAMPLE);
    else
        t_cross = i*SAMPLE;

    /*      user abort?                    */
    if ( char_at_keybd == 's' )
        fprintf(stderr,"user killed the program\n");

    fprintf(stderr,"return to see results ->\n");
    gets(string);

    /*      output results                 */

    fprintf(out_put,"sample= %4.2f     ",SAMPLE);
    fprintf(out_put,"t opt radius to 0.0");
    fprintf(out_put,"= %.7e\n",TOPTCIRCL);
    fprintf(out_put,"x0, x1, u, t\n");
    for( j=0; j <= i; j++ ) {
        fprintf(out_put,"%10f %10f %10f %10f\n"
                        ,b0[j],b1[j],b2[j],j*SAMPLE);
    }
    fprintf(out_put,"\noptimal time to radius =%f\n",
                   (t_final_theo-TOPTCIRCL));
    fprintf(out_put,"approximate actual time to radius");
    fprintf(out_put," = %f\n",t_cross);

    fprintf(stderr,"\noptimal time to radius =%f\n",
                   (t_final_theo-TOPTCIRCL));
    fprintf(stderr,"approximate actual time to radius ");
    fprintf(stderr,"= %f\n",t_cross);

    /*      free history buffer            */
    free(b0);
    free(b1);
    free(b2);

    fprintf(stderr,"comm_bit= %d\nexiting....\n",comm_bit);
    exit();
}
```

```
/*********************************************************
*
*   INTERSECTION -- calculates exact time crossed exit
*   criterion -- a circle of radius (sample rate)**2/4.0.
*
*   intersection( x00, x10, u, time1)
*
*   intersection calculates the exact time the path
*   crossed the exit criterion.
*       x00 = position outside exit circle.
*       x10 = position inside exit circle.
*       u = maximum/minimum control.
*       time1 = sample time at position x00.
*
*********************************************************/

float intersection( x00, x10, u, time1)
float x00, x10;
float u, time1;
{
    double k, x0_i, x1_i;
    float time_i;
    double work1, work2, work3;

    k = x00 - x10*x10/2.0/u;
    work1 = u*u + 2.0*u*k + 0.06*0.06;
    work1 = sqrt( work1 );
    work2 = -u + work1;
    work3 = -u - work1;
    x0_i = (absval(work2) <= absval(work3) ? work2 : work3 );
    x1_i = sqrt( (double) (2*u*(x0_i - k) ) );
    time_i = time1 + absval( (absval(x10) - x1_i) );
    return(time_i);
}


/*********************************************************
*
*   FREQ -- interrupt routine
*   occurs at sample times.
*
*********************************************************/

void freq(pregs, pisrblk, pmsg)
ALLREG *pregs;
ISRCTRL *pisrblk;
ISRMSG *pmsg;
{
    extern int comm_bit;
    static int first_time = 1;
```

```
    /*      send an end-of-interrupt to controller  */
    utintoff();
    eoi_int();
    utinton();

    /*      comm_bit == 0
                sample time, send over vector
            comm_bit > 0
                error, did not make it from last sample
                time                                      */

    if ( comm_bit == 0 && !first_time) {
        send();
        comm_bit = 1;
    } else if (first_time) {
        first_time = 0;
    } else {
        comm_bit = 5;
    }
}


/************************************************************
*
*   TERMINAL_COUNT -- interrupt routine
*   occurs when set number of DMA transmissions occurred.
*
************************************************************/

void terminal_count(pregs, pisrblk, pmsg)
ALLREG *pregs;
ISRCTRL *pisrblk;
ISRMSG *pmsg;
{
    extern int comm_bit;
    extern float x[];

    static float t, dx[2];
    static float q[2];

    int neq;
    float h;

    /*      send an end-of-interrupt to controller  */
    utintoff();
    eoi_int();
    utinton();

    /*      number of first order equations */
    neq = 2;
```

```
    /*      time step                        */
    h = SAMPLE;

    /*      turn off transmit/receive signal    */
    auxloff();

    /*      comm_bit == 1
                just sent vector out
            comm_bit == 2
                just received vector, integrate to
                next step
            comm_bit > 2
                error                                  */

    if ( comm_bit == 1 ) {
        receiv();
        comm_bit = 2;
    }
    else if ( comm_bit == 2 ) {
        comm_bit = 3;
        /*      integrate to next step  */
        rkg( neq, h, t, x, dx, q);
        /*      update time             */
        t += h;
    } else {
        comm_bit = 6;   /* error code   */
    }
}
```

```
/*********************************************************
*
*   ZRSIMUL -- simulation program for the Zenith
*
*   zrsimul set up a simulation of a digital control system
*   with the starting positions generated by a pseudo-random
*   generator.
*   Two interrupt routines are installed.  1) FREQ is the
*   interrupt generated by the sample rate, and
*   2) TERMINAL_COUNT is the interrupt generated each time
*   the specified number of DMA transmissions has occurred.
*   The external line AUX 1 is used to signal when the
*   computer is ready to transmit or receive.  AUX 2 is
*   used to enable the sample frequency.
*
*   AUTHOR:     Donald A. Smith
*   DATE:       8/5/88
*
*********************************************************/

#define LINT_ARGS
#include <utility.h>
#include <stdlib.h>
#include <bisr.h>
#include <dos.h>
#include <malloc.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include "loc.h"

float x[10];          /* x[0]=state variable x0
                         x[1]=state variable x1
                         x[2]=control u          */
int comm_bit = 0;     /* test variable to tell when we have
                         been interupted                  */

/* FUNCION DEFINITIONS       */
void freq(ALLREG *, ISRGTRL *, ISRMSG *);
void terminal_count(ALLREG *, ISRGTRL *, ISRMSG *);
float intersection( float, float, float, float);

main()
{
    /*      PDMA base address       */
    int base_add = 0x300;
    /*      DMA level               */
    int dma_level = 1;
    /*      hardware interrupt #5 for 'terminal_count'  */
    int tc_level = 5;
```

```
/*      hardware interrupt #4 for 'freq'    */
int freq_level = 4;
/*      source of 'terminal_count' interrupt    */
int tc_sourc = 2;
/*      source of 'freq' interrupt  */
int freq_sourc = 4;
/*      number of DMA transmissions */
unsigned numb_trans = 6;
/*      transmit words instead of bytes */
unsigned length = 1;
/*      initialize to output    */
unsigned direction = 1;
/*      enable DMA recycle      */
unsigned recycle = 1;
/*      transmit request external   */
unsigned trans_sourc = 0;
unsigned data_segment;

unsigned char work1,work2;
unsigned int work3;

struct SREGS segregs;
FILE *out_file;
char file_name[16];

float t_final_theo, switch_pt;
double p0, p1;
float x00, x10, x01, x11, x0_prev, x1_prev;

extern float x[];
extern int comm_bit;

unsigned int i,j,k;
int iterations, start_pt;
char char_at_keybd;
char string[80];

/*      pointers to history array   */
float hist_of[5][1001];

char_at_keybd = '!';  /* initilize to some value */

/*      disable DMA mode 1          */
outpt(0x0a,0x05);
/*      hardware check              */
if ( mode0(base_add, dma_level, tc_level))
    fprintf(stderr,"error on termination of mode0");

/*      set hardware communication lin off  */
aux1off();
```

```
/*      disable sampling frequency from the HP  */
aux2off();  /* disable the hp's counter */

/*      request user for number of iterations   */
do {
    fprintf(stderr,"number of random iterations?\n");
    gets(string);
    iterations = atoi(string);
    fprintf(stderr,"starting point?\n");
    gets(string);
    start_pt = atoi(string);
    if ( iterations > 1000 || start_pt > 999 )
        fprintf(stderr,
            "max # of iteratations is 1000\n");
} while ( iterations > 1000 || start_pt > 999 );

/*      request user for output file    */
fprintf(stderr,"file to output to?\n");
gets(file_name);
out_file = fopen(file_name,"w");

fprintf(stderr,"\nsample rate= %4.2f\n\n",SAMPLE);
fprintf(stderr,"return to continue ->\n");
gets(string);

utintoff();
/*      install 'terminal_count interrupt   */
mode7( base_add, tc_level, tc_sourc, terminal_count, 1);
/*      install 'freq' interrupt            */
mode7( base_add, freq_level, freq_sourc, freq, 1);
/*      set output frequency for transmission rate  */
mode3( base_add, TRANS1_DIV, TRANS2_DIV);
utinton();

/*      get x() segment         */
segread( &segregs );
data_segment = segregs.ds;
/*      set up DMA              */
mode1(base_add, dma_level, numb_trans, length,
    direction, recycle, trans_sourc, data_segment, x);


for( k = 0; k < iterations; k++ ){
    /*  generate random number  */
    /* 5461.1667 = +-6  */
    x[0] = rand()/5461.1667;
    x[1] = rand()/5461.1667;
    /* was 5462.  */
    x[0] = ( rand() < 16383 ) ?  x[0] : -x[0];
    x[1] = ( rand() < 16383 ) ?  x[1] : -x[1];
```

```
/*  save the number */
hist_of[0][k] = x[0];
hist_of[1][k] = x[1];
comm_bit = 0;
/*  calculate optimal if at starting point  */
if( k >= start_pt) {
    fprintf(stderr,"%4d %10f %10f ",k,x[0],x[1]);
    switch_pt = x[0]+x[1]*absval(x[1])*0.5/UMAX;
    if( switch_pt < 0.0 ) {
        p0 = -1.0/(sqrt( (double)(0.5*x[1]*x[1] -
            UMAX*x[0])));
        p1 = ( -1.0 - p0*x[1] )/UMAX;
        t_final_theo = 2.0*p0*x[0] + p1*x[1];

     } else if( switch_pt > 0.0 ) {
        p0 = 1.0/(sqrt( (double)(0.5*x[1]*x[1] +
            UMAX*x[0])));
        p1 = ( 1.0 + p0*x[1] )/UMAX;
        t_final_theo = 2.0*p0*x[0] + p1*x[1];

    } else {
        t_final_theo = x[1];
    }

    /*  start of simulation loop    */
    for( i=1; ; i++ ) {

        x0_prev = x[0];
        x1_prev = x[1];

        /*      first time start sample rate    */
        if ( i == 1 )
            aux2on();

        /*      wait for sample interval or abort */
        while( comm_bit == 0 ) {
            if ( key_push() ) {
                char_at_keybd = getch();
                if ( char_at_keybd == 's' )
                    /* exit the simulation */
                    goto kickout;
            }
        }

        /*      turn on line ready to send vector */
        aux1on();

        /*      wait until position sent out
                or abort                        */
        while( comm_bit == 1 ) {
```

```
            if ( key_push() ) {
                char_at_keybd = getch();
               if ( char_at_keybd == 's' )
                    /* exit the simulation */
                    goto kickout;
            }
        }

        /*      turn on line to receive vector  */
        auxlon();

        /*      wait until vector has been recieved
                or abort          */
        while( comm_bit == 2 ) {
            if ( key_push() ) {
                char_at_keybd = getch();
                if ( char_at_keybd == 's' )
                    /* exit the simulation */
                    goto kickout;
            }
        }

        /*      if error abort       */
        if ( comm_bit != 3 ) {
            fprintf(stderr,"error, comm_bit=%d\n",
                comm_bit);
            goto kickout;
        }

        /*      check exit criterion    */
        if ( x[0]*x[0] + x[1]*x[1] < ( 0.06*0.06)) {

            x00 = x0_prev;
            x10 = x1_prev;
            x01 = x[0];
            x11 = x[1];
            break;
        }

        comm_bit = 0;
    }

    /*  turn off sampling frequency     */
    aux2off();

    /*      save in history list    */
    hist_of[2][k] = i*SAMPLE;
    hist_of[3][k] = t_final_theo;

    if( x[2] == 0.0 ) {
```

```
                fprintf(stderr,
                    "u was 0.0 before intersec\n");
                x[2] = UMAX;
            }

            hist_of[4][k] = intersection(x00, x10, x[2],
                hist_of[2][k]-SAMPLE);

            if( (hist_of[3][k] - TOPTCIRCL) == 0.0 )
                fprintf(stderr,
                    "TOPT subtraction was zero, %f %f\n",
                    hist_of[3][k],TOPTCIRCL);

            fprintf(stderr,"%10f %10f %10f %10f\n",
                hist_of[4][k],hist_of[2][k], hist_of[3][k] -
                TOPTCIRCL,hist_of[4][k]/(hist_of[3][k]
                    - TOPTCIRCL) );
        } else {
            hist_of[2][k] = hist_of[3][k] = hist_of[4][k]
                = 0.0;
        }
    }

kickout:        /* exit the simulaton  */
    /*      turn off DMA mode 1     */
    outpt(0x0a,0x05);
    /*      turn off sampling frequency     */
    aux2off();
    utintoff();
    /*      uninstall interrupts    */
    mode7( base_add, tc_level, tc_sourc, terminal_count, 0);
    mode7( base_add, freq_level, freq_sourc, freq, 0);
    utinton();

    if ( char_at_keybd == 's' ) {
        fprintf(stderr,"user killed the simulation\n");
        fprintf(out_file,"user killed the simulation\n");
    }

    if ( comm_bit == 5 || comm_bit == 6 )
        fprintf(out_file,"error, comm_bit=%d\n",comm_bit);

    /* output results       */
    fprintf(out_file,"sample= %4.2f     ",SAMPLE);
    fprintf(out_file,"t opt radius to 0.0= %.7e\n",
        TOPTCIRCL);
    fprintf(out_file,"x[0],x[1],t cross,"
    fprintf(out_file," t final actual, t opt to radius");
    fprintf(out_file,", t cross/t opt to rad\n");
    for ( i=start_pt; i < k; i++) {
```

```
        fprintf(out_file,"%4d %10f %10f %10f %10f %10f %10f\n"
          ,i,hist_of[0][i],hist_of[1][i],hist_of[4][i],
          hist_of[2][i], hist_of[3][i] - TOPTCIRCL,
          hist_of[4][i]/(hist_of[3][i] - TOPTCIRCL) );
    }
    exit();
}

/************************************************************
*   INTERSECTION -- calculates exact time crossed exit
*   criterion -- a circle of radius 0.06.
*
*   intersection( x00, x10, u, time1)
*
*       x00 = position outside exit circle
*       x01 = position inside exit circle
*       u = maximum/minimum control
*       time1 = sample time a position x00.
*
************************************************************

float intersection( x00, x10, u, time1)
float x00, x10;
float u, time1;
{
    double k, x0_i, x1_i;
    float time_i;
    double work1, work2, work3;

    k = x00 - x10*x10/2.0/u;
    work1 = u*u + 2.0*u*k + 0.06*0.06;
    if( work1 < 0.0 ) {
        fprintf(stderr,"0 in intersec, sqrt of negative,");
        fprintf(stderr," %f %f %f\n",work1,u,k);
        work1 = -work1;
    }
    work1 = sqrt( work1 );
    work2 = -u + work1;
    work3 = -u - work1;
    x0_i = (absval(work2) <= absval(work3) ?
        work2 : work3 );
    work1 = 2.0*u*(x0_i - k);
    if( work1 < 0.0) {
        fprintf(stderr,"1 in intersect, sqrt of negative,");
        fprintf(stderr," %f %f %f %f\n",work1,u,x0_i,k);
        work1 = -work1;
    }
    x1_i = sqrt( work1 );
    time_i = time1 + absval( (absval(x10) - x1_i) );
    return(time_i);
```

```
}

/**********************************************************
*
*   FREQ -- interrupt routine
*   occurs at sample times
*
**********************************************************/

void freq(pregs, pisrblk, pmsg)
ALLREG *pregs;
ISRCTRL *pisrblk;
ISRMSG *pmsg;
{
    extern int comm_bit;
    static int first_time = 1;

    /*      send an end-of-interrupt to controller  */
    utintoff();
    eoi_int();
    utinton();

    /*      comm_bit == 0
                sample time, send over vector
            comm_bit > 0
                error, did not make it from last sample .
                time        */

    if ( comm_bit == 0 && !first_time) {
        send();
        comm_bit = 1;
    } else if (first_time) {
        first_time = 0;
    } else {
        comm_bit = 5;
    }
}

/**********************************************************
*
*   TERMINAL_COUNT -- interrupt routine
*   occurs when set number of DMA trasnmissions occured.
*
**********************************************************/

void terminal_count(pregs, pisrblk, pmsg)
ALLREG *pregs;
ISRCTRL *pisrblk;
ISRMSG *pmsg;
{
```

```
    extern int comm_bit;
    extern float x[];

    static float t, dx[2];
    static float q[2];

    int neq;
    float h;

    /*      send an end-of-interrupt to controller  */
    utintoff();
    eoi_int();
    utinton();

    /*      number of first order equations         */
    neq = 2;
    h = SAMPLE;

    /*      turn off transmit/receive signal        */
    auxloff();

    /*      comm_bit == 1
                just sent vector out
            comm_bit == 2
                just received vector, intergrate to
                next step
            comm_bit > 2
                error                               */

    if ( comm_bit == 1 ) {
        receiv();
        comm_bit = 2;
    }
    else if ( comm_bit == 2 ) {
        /* control is over, integrate it */
        comm_bit = 3;
        rkg( neq, h, t, x, dx, q);
        t += h;
    } else {
        comm_bit = 6;           /* error code */
    }
}
```

```
/************************************************************
*
*   DERIV -- calculates derivatives for a double integrator
*
*   deriv( neq, t, x, dx)
*
*
*   deriv calculates the derivatives based on the double
*   integrator problem.
*           dx1 = x2
*           dx2 = u (the control)
*       neq = number of first order equations.
*       t = the independent variable.
*       x(i) = the dependent variable:
*               x = { x1, x2, u }.
*       dx(i) = derivative of the dependent variable.
*
*   AUTHOR:      Donald A. Smith
*   DATE:        6/21/88
*
************************************************************/


deriv( neq, t, x, dx)

/* external functions called
none
*/

int neq;
float t;
float x[];
float dx[];

{
    dx[0] = x[1];
    dx[1] = x[2];

}
```

```
/*************************************************************
*
*   RKG -- Runga-Kutta-Gill integration
*
*   rkg( neq, h, x, y, dy, q)
*
*   rkg integrates a set of first order differential
*   equations.
*       Y(i) = dependent varible.
*       DY(i) = derivative of dependent variable.
*       neq = number of first order equations.
*       h = interval size for integration.
*       q(neq) = work array.
*
*   AUTHOR:     Donald A. Smith
*   DATA:       6/21/88
*
*************************************************************/


rkg( neq, h, x, y, dy, q)

/*  external functions called
deriv(neq, x, y, dy) - calculates the derivatives
*/

int neq;
float h;
float x;
float y[];
float dy[];
float q[];
{
    /*     integration constants            */
    float a[2];
    float h2,b;
    int i,j;
    a[0] = 0.292893218813452;
    a[1] = 1.7071067811865471;
    /*      zero out the work array         */
    for ( i = 0; i < neq; i++)              .
        q[i] = 0.0;
    h2 = 0.5*h;
    /*      get derivatives                 */
    deriv( neq, x, y, dy);
    for ( i=0; i < neq; i++) {
        b = h2*dy[i]-q[i];
        y[i] += b;
        q[i] += 3.0*b-h2*dy[i];
    }
```

```
    x += h2;
    for ( j = 0; j < 2; j++) {
        deriv( neq, x, y, dy);
        for ( i = 0; i < neq; i++) {
            b = a[j]*(h*dy[i]-q[i]);
            y[i] += b;
            q[i] += 3.0*b-a[j]*h*dy[i];
        }
    }
    x += h2;
    deriv( neq, x, y, dy);
    for ( i = 0; i < neq; i++) {
        b = 0.166666666666*(h*dy[i]-2.0*q[i]);
        y[i] += b;
        q[i] += 3.0*b-h2*dy[i];
    }

    return(0);
}
```

HPSIMUL - Switching Curves Control

HPSIMUL is the double integrator controller program. HPSIMUL is a small program and call only small model format subroutines. It uses the true switching curves to calculate control.

```
/*********************************************************
*
*   HPSIMUL -- simulation program for Hewlett Packard
*              switching curve control scheme
*
*   hpsimul sets up a controller environment for the HP
*   computer.  It calls routines to install the interrupts
*   and initialize the DMA controller.
*   One interrupt routine is installed  - 'terminal_count'.
*   this interrupt is generated each time the specified
*   number of DMA transmissions has occurred.  The external
*   line AUX1 is used to signal the hand-shaking hardware
*   that the computer is ready to transmit or receive.
*   To gracefully exit press the 's' key.
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/21/88
*
*********************************************************/

#define LINT_ARGS
#include <utility.h>
#include <stdlib.h>
#include <bisr.h>
#include <dos.h>
#include <malloc.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include "loc.h"


float x[10];                /* x[0] = state variable x1
                               x[1] = state variable x2
                               x[2] = control u          */
int comm_bit = 0;           /* test variable for communication
                               with the interrupt routines     */

/* FUNCTION DEFINITIONS    */
/* interrupt routine for terminal count*/
void terminal_count( ALLREG *, ISRCTRL *, ISRMSG *);

main()
{
    /*      base address of PDMA board  */
    int base_add = 0x300;
    /*      DMA level                   */
    int dma_level = 1;
    /*      hardware interrupt for terminal_count */
    int tc_level = 5;
```

```
/*      source of terminal_count interrupt */
int tc_sourc = 2;
/*      number of DMA transmissions */
unsigned numb_trans = 6;
/*      transmit words instead of bytes  */
unsigned length = 1;
/*      initialize direction for input */
unsigned direction = 0;
/*      enable DMA recycle bit          */
unsigned recycle = 1;
/*      transmit request external */
unsigned trans_sourc = 0;
unsigned data_segment;

struct SREGS segregs;

extern float x[];
extern int comm_bit;

int i, j;
char char_at_keybd;
char string[80];

char_at_keybd = '!';   /* initialize to something */

/*      disable DMA on mode 1 in case was left on */
outpt(0x0a,0x05);

/** hardware check */
if ( mode0(base_add, dma_level, tc_level) ) {
    fprintf(stderr,"error with mode 0, check hardware");
    exit();
}

/*      set hardware communication line off */
auxloff();

fprintf(stderr,"\nsample rate= %4.2f\n\n",SAMPLE);
fprintf(stderr,"return to continue ->\n");
gets(string);

/*      install interrupt routine 'terminal_count' */
utintoff();
mode7( base_add, tc_level, tc_sourc, terminal_count, 1);
/*      set output frequency of timer for sample rate */
mode3( base_add, MULT_1, MULT_2 );
utinton();

/*      get x() segment              */
segread( &segregs );
```

```
data_segment = segregs.ds;
/*      set up DMA                    */
model( base_add, dma_level, numb_trans, length,
       direction, recycle, trans_sourc,
       data_segment, x);

/*      initialize to receive data  */
receiv();

comm_bit=0;
fprintf(stderr,"start\n");


/*** start simulation loop        */

for( i=0; ; i++) {
    /*      turn on line signaling ready to receive */
    auxlon();

    /*      wait until vector has arrived or abort */
    while( comm_bit == 0 ) {
        if ( key_push() ) {
            char_at_keybd = getch();
            if ( char_at_keybd == 's' )
                goto kickout; /* exit the simulation */
        }
    }

     /*      received vector and calculated control
            ready to send back vector                 */
    auxlon();

    /*      wait until vector has been
            sent out or user aborts                   */
    while( comm_bit == 1 ) {
        if ( key_push() ) {
            char_at_keybd = getch();
            if ( char_at_keybd == 's' )
                goto kickout; /* exit the simulation */
        }
    }

    /*      check if error condition occurred   */
    if ( comm_bit == 5 ) {
        fprintf(stderr,"error code, comm_bit=%d\n",
                comm_bit);
        break;
    }

}
```

```
/*** exit simulation                    */
kickout:
    /*      turn off DMA                */
    outpt(0x0a,0x05);

    /*      uninstall interrupt and set frequency to 0 */
    utintoff();
    mode7( base_add, tc_level, tc_sourc, terminal_count, 0);
    utinton();
    fprintf(stderr,"interrupts changed back\n");
    mode3( base_add, 0, 0);

    /*      user abort?                 */
    if ( char_at_keybd == 's')
    fprintf(stderr,"user killed the simulation\n");

    fprintf(stderr,"exiting.... \n");
    exit();
}
```

```
/************************************************************
*
*   TERMINAL_COUNT -- interrupt routine
*
*   occurs when set number of DMA transmissions has occurred.
*
************************************************************/


void terminal_count( pregs, pisrblk, pmsg)

ALLREG *pregs;
ISRCTRL *pisrblk;
ISRMSG *pmsg;

{
    extern float x[];
    extern int comm_bit;

    /*      send an end-of-interrupt to controller  */
    utintoff();
    eoi_int();
    utinton();

    /*      turn off transmit/receive signal    */
    auxloff();

    /*      comm_bit = 0
                just received vector, reverse direction,
                calculate control, and update communication
            comm_bit = 1
                just sent back vector, reverse
                direction, and update communication
            comm_bit > 1
                error, set to error condition and return
                                                        */

    if ( comm_bit == 0) {
        send();
        x[2] = control( x, (float) UMAX);
        comm_bit = 1;
    } else if ( comm_bit == 1) {
        receiv();
        comm_bit = 0;
    } else {
        comm_bit = 5;
    }
}
```

```
/************************************************************
*
*   CONTROL -- calculate control based upon switching curves
*
*   float control( x, umax)
*
*
*   control calculates the control based upon the switching
*   curves of the double integrator problem.
*       x = array of x1,x2,u.
*       umax = maximum limit of u for bang-bang control.
*   returns a float ( control ).
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/21/88
*
************************************************************/

#define absval(a) ( ( a > 0 ) ? a : -a )
#include <stdio.h>

/* external fuctions called
none
*/

float control( x, umax)

float x[];
float umax;

{
    /*      parameter for location on phase plane   */
    float switch_curv;

    switch_curv = x[0]+x[1]*absval(x[1])*0.5/umax;

   if ( switch_curv > 0. ) {

        return( -umax);

   } else {

        return( umax );
   }


}
```

HPSIMUL Finite Elements Control

This form of HPSIMUL uses the R-Theta finite element grid of isochrones to calculate control. It uses the large model format and calls only large model format subroutines.

```
/**********************************************************
*
*   SIMULATION HEADER FILE
*
*   information in this header is used by HPSIMUL
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/22/88
*
**********************************************************/

#define LINT_ARGS
/*      S_RATE = 1  -- sample rate 0.01
               = 2  -- sample rate 0.02
               = 3  -- sample rate 0.03     */
#define S_RATE  3

#define UMAX 1.0

/* time from termination circle to the origin  */
#define TOPTCIRCL 5.9973043E-2

#if S_RATE == 1
    #define SAMPLE     0.01
    #define MULT_1     1000
    #define MULT_2     100
#elif S_RATE == 2
    #define SAMPLE     0.02
    #define MULT_1     1000
    #define MULT_2     200
#elif S_RATE == 3
    #define SAMPLE     0.03
    #define MULT_1     1000
    #define MULT_2     300
#elif S_RATE == 4
    #define SAMPLE     0.5
    #define MULT_1     1000
    #define MULT_2     5000
#endif

#define absval(a)  ( ( (a) > 0 ) ? (a) : -(a) )

/*      function definitions    */
void eoi_int(void);

void aux1on(void);
void aux1off(void);
void aux2on(void);
void aux2off(void);
void aux3on(void);
```

```
void aux3off();

void send(void);
void receiv(void);

void rkg( int, float, float, float *, float *, float *);
float control( float *, float);

int key_push(void);

int nodes(int, int, int *);
int gridlt( int, int, int *);
int read_in_nodes( void );
void fi_driver(void);

/* SCALEF is the the outer radius */
#define SCALEF 28.5

/* NUMSPKS is number of spokes */
#define NUMSPKS  15

/* NUMCIR is the number of circles */
#define NUMCIR   9

/* GRIDF is the grid factor for scaling */
#define GRIDF    1.8

/* buffer size for nodes and elements */
#define BUF_NODE   1700
#define BUF_ELEM   3600
```

```
/**********************************************************
 *
 *   HPSIMUL -- simulation program for Hewlett Packard.
 *              finite element control scheme
 *
 *   hpsimul sets up a controller environment for the HP
 *   computer.  It calls routines to install the interrupts
 *   and initialize the DMA controller.
 *   One interrupt routine is installed - 'int_h_asm'.
 *   This interrupt is generated each time the specified
 *   number of DMA transmissions has occurred.  This
 *   interrupt routine is an assembly program that passes
 *   control to the 'C' program 'my_int_hand' which
 *   calculates the new control.  The external line
 *   AUX1 is used to signal the hand_shaking hardware
 *   that the computer is ready to transmit or receive.
 *   To gracefully exit press the 's' key.
 *
 *   AUTHOR:      Donald A. Smith
 *   DATE:        6/22/88
 *
 **********************************************************/

#include <utility.h>
#include <stdlib.h>
#include <bisr.h>
#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include "loc.h"

float x[10];          /* x[0] - state variable x1
                         x[1] - state variable x2
                         x[2] - control u          */
int comm_bit = 0;     /* test variable for communication
                         with the interrupt routines      */

/*      x and y locations of each node  */
float x_node[BUF_NODE], y_node[BUF_NODE];
/*      node table                      */
int ntable[BUF_ELEM][3];
/*      solution vector of final times for each node    */
float tf[BUF_NODE];
int ib[BUF_ELEM];

/*      FUNCTION DECLARATIONS                  */
void my_int_hand(void);
void int_h_asm(void);
```

```
main()
{
    /*      base address of PDMA board  */
    int base_add = 0x300;
    /*      DMA level                   */
    int dma_level = 1;
    /*      hardware interrupt for int_h_asm    */
    int tc_level = 5;
    /*      source of int_h_asm interrupt   */
    int tc_sourc = 2;
    /*      number of DMA transmissions */
    unsigned numb_trans = 6;
    /*      transmit words instead of bytes   */
    unsigned length = 1;
    /*      initialize direction for input  */
    unsigned direction = 0;
    /*      enable DMA recycle          */
    unsigned recycle = 1;
    /*      transmit signal external */
    unsigned trans_sourc = 0;
    unsigned data_segment, x_offset;
    long int x_work;

    struct SREGS segregs;

    extern float x[];
    extern int comm_bit;

    int i, j;
    char char_at_keybd;
    char string[80];

    char_at_keybd = '!';    /* initialize to something */

    /*      disable DMA on mode 1 in case was left on    */
    outpt(0x0a,0x05);
    /** hardware check  */

    if ( mode0(base_add, dma_level, tc_level) )
        fprintf(stderr,"error with mode 0, check hardware");
    fprintf(stderr,"mode0\n");

    /*      set hardware communication line off */
    auxloff();

    fprintf(stderr,"\nsample rate= %4.2f\n\n",SAMPLE);

    /*      initialize finite element grid --
                node table, nodal position, and final time*/
```

```
fi_driver();

fprintf(stderr,"return to continue ->\n");
gets(string);


utintoff();

/*      install interrupt routine int_h_asm */
mode8( base_add, tc_level, tc_sourc, int_h_asm, 1);

/*      set output frequency of time for sample rate */
mode3( base_add, MULT_1, MULT_2 );

utinton();

/*      break long pointer for x() into offset and
        segment                                       */
x_work = x;
data_segment = (unsigned)(x_work >> 16);
x_offset = (unsigned)(x_work);

/*      set up DMA                    */
model( base_add, dma_level, numb_trans, length,
       direction, recycle, trans_sourc, data_segment,
       x_offset);

/*      initialize to receive data  */

receiv();

comm_bit= 0;

fprintf(stderr,"start\n");

/*** start simulation loop       */

for( i=0; ; i++) {
    /*      turn on line signaling ready to receive */
    auxlon();

    /*      wait until vector has arrived or abort  */
    while( comm_bit == 0 ) {
        if ( key_push() ) {
            char_at_keybd = getch();
            if ( char_at_keybd == 's' )
                goto kickout; /* exit the simulation */
        }
    }
```

```
        /*      received vector and calculated control
                ready to send back vector                   */

        auxlon();

        /*      wait until vector has been sent out or
                user aborts                                 */
        while( comm_bit == 1 ) {
            if ( key_push() ) {
                char_at_keybd = getch();
                if ( char_at_keybd == 's' )
                    goto kickout; /* exit the simulation */
            }
        }

        /*      check if error condition occurred   */
        if ( comm_bit == 5 ) {
            fprintf(stderr,"error code, comm_bit=%d\n",
                comm_bit);
            goto kickout; /* exit the simulation    */
        }
    }

    /*** exit simulation                          */
kickout:
    /*      turn off DMA                       */
    outpt(0x0a,0x05);

    /*      uninstall interrupt and set frequency to 0  */
    utintoff();
    mode8( base_add, tc_level, tc_sourc, int_h_asm, 0);
    mode3( base_add, 0, 0);
    utinton();

    if ( char_at_keybd == 's')
    fprintf(stderr,"user killed the simulation\n");
    fprintf(stderr,"exiting.... \n");

    exit();
}
```

```
/*************************************************************
*
*   CONTROL -- calculates control for double integrator
*   using finite element isochrones
*
*   control( x, umax)
*
*   control calculates the control for the current position
*   for bang bang control for a double integrator.  The
*   control is calculated using the isochrones.  The
*   isochrones are generated by using the final times
*   for each node in the finite element mesh.
*       x = array of (x1, x2, u).
*       umax = scale for max/min control.
*   EXTERNAL VARIABLES:
*       x_node, y_node = x and y position of each node.
*       ntable = node table.
*       tf = final time vector for nodes.
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/22/88
*
*************************************************************/

#include "loc.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*      signed control  */
#define sgncont(a)  ((a) > 0 ? umax : -umax )

float control(x, umax)

/* external functions called
none
*/

float *x;
float umax;

{
    extern float x_node[], y_node[];
    extern int ntable[][3];
    extern float tf[];

    float x_0, x_1;

    /*      element number  */
    int el_num;
```

```
float rad_test, rad, d_pow;
int min, max, i, j, k;

int start_node, spknum;
float a, b, c, d, theta;

int nt_0, nt_1, nt_2;

float slope_1, yints_1, slope_2, x_i, y_i;

float b_1, c_1, c_2, c_3, a_1, del2, lamda, u;

/*      which element is current position in?    */

x_0 = x[0];
x_1 = x[1];

/*      divide by zero elemination */
if( x_0 == 0.0 )
    x_0 = 0.00001;
if( x_1 == 0.0 )
    x_1 = 0.00001;

/*      radius to current position */
rad = x_0*x_0 + x_1*x_1;
rad = sqrt( (double) rad);

/*      find the circle current position is in */
for( min=0, max=NUMCIR, j=1; j<4; j++) {
    i = min + (max-min)/2;

    for( k=NUMCIR-i, d_pow=1.0; k>0; k--) {
        d_pow *= GRIDF;
    }
    rad_test = SCALEF/d_pow;

    if ( rad < rad_test)
        max = i;
    else
        min = i;
}

for( min++; ; min++) {
    for( k=NUMCIR-min, d_pow=1.0; k>0; k--) {
        d_pow *= GRIDF;
    }
    rad_test = SCALEF/d_pow;
    if ( rad_test > rad) {
        min--;
        break;
```

```
        }
    }

    if( min == 0) {
        el_num = 1;
        start_node = 2;
    } else {
        el_num = NUMSPKS + (min-1)*2*NUMSPKS;
        start_node = min*NUMSPKS + 2;
    }

    a = x_node[start_node];
    b = y_node[start_node];
    rad_test = a*a + b*b;
    rad_test = sqrt( (double) rad_test);
    a = a/rad_test;
    b = b/rad_test;
    c = x_0/rad;
    d = x_1/rad;

    theta = atan2( (double)(a*d-c*b), (double)(a*c+b*d) );
    if ( theta < 0.0 )
        theta += 6.28319;
    spknum = (theta/6.28319)*NUMSPKS;

    if( min > 0) {

        el_num += spknum*2;

        nt_0 = ntable[el_num+1][0];
        nt_1 = ntable[el_num+1][1];
        nt_2 = ntable[el_num+1][2];

        if( spknum < NUMSPKS - 1)
            slope_1 = (y_node[ nt_0 ] - y_node[ nt_1 ]) /
                      (x_node[ nt_0 ] - x_node[ nt_1 ]);
        else
            slope_1 = (y_node[ nt_0 ] - y_node[ nt_2 ]) /
                      (x_node[ nt_0 ] - x_node[ nt_2 ]);

        yints_1 = y_node[ nt_0 ] -
                  slope_1*x_node[ nt_0 ];
        slope_2 = x_1/x_0;
        x_i = yints_1/( slope_2 - slope_1);
        y_i = slope_2 * x_i;
        rad_test = x_i*x_i + y_i*y_i;
        if ( rad_test > rad*rad )
            el_num += 1;
        else
            el_num += 2;
```

```
    } else
        el_num += spknum + 1;

    nt_0 = ntable[el_num][0];
    nt_1 = ntable[el_num][1];
    nt_2 = ntable[el_num][2];

    b_1 = y_node[ nt_1 ] - y_node[ nt_2 ];
    c_1 = x_node[ nt_2 ] - x_node[ nt_1 ];
    c_2 = x_node[ nt_0 ] - x_node[ nt_2 ];
    c_3 = x_node[ nt_1 ] - x_node[ nt_0 ];
    a_1 = x_node[ nt_1 ]*y_node[ nt_2 ] -
          x_node[ nt_2 ]*y_node[ nt_1 ];
    del2 = a_1 + b_1*x_node[ nt_0 ] +
                 c_1*y_node[ nt_0 ];
    lamda = (c_1*tf[ nt_0 ] + c_2*tf[ nt_1 ] +
             c_3*tf[ nt_2 ]) / del2;
    u = sgncont( -lamda );

    return( u );

}
```

```
;**********************************************************
;*
;*      INT_H_AS -- interrupt routine
;*
;*      occurs when set number of DMA transmissions has
;*      occurred.  Passes control to 'G' program
;*      'my_int_hand'.
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/22/88
;*
;**********************************************************
;
;
.286c                                   ;compile for 80286
.287                                    ;compile for 80287
EOI_PORT EQU    20H                     ;port address of the
                                        ;controller
EOI_CMD  EQU    20H                     ;interrupt acknowledge
                                        ;command
INT_DATA  segment word public 'DATA'
save_ss dw      3412H                   ;old stack segment
save_sp dw      0AA55H                  ;old stack pointer
istack  dw 512 dup (0)                  ;new stack space
topstk  dw      0                       ;top of the stack
astack  dw      topstk                  ;save address of stack
env_cop db 94 dup (?)                   ;space for 80287
                                        ; environment
INT_DATA ends
;
DGROUP  GROUP   INT_DATA
;
;
EXTRN   _my_int_hand:FAR        ;c program to be called
intrp_TEXT  segment para public 'CODE' ;define as code
        ASSUME  cs:intrp_TEXT, ds:DGROUP, es:DGROUP
        public _int_h_asm
        _int_h_asm      proc    far         ;far procedure
        sti
        fwait                           ;wait for 80287 to catch
                                        ; up
        cli
        push    ax                      ;push the registers
        push    bx                      ;to save the current
        push    cx                      ;state
        push    dx
        push    si
        push    di
        push    bp
        push    ds
```

```
        push    es
;
;
        mov     ax,SEG DGROUP       ;segment of data
        mov     es,ax
;
        mov     es:word ptr save_ss,ss ;save stack seg.
        mov     es:word ptr save_sp,sp ;save stack ptr.
        mov     ds,ax                  ;change to the new
        mov     ss,ax                  ; stack
        mov     sp,OFFSET es:astack
;
        sti                         ;allow more interrupts
        FNSAVE  env_cop             ;save environment of
                                    ; the 80287
        mov     bp,sp               ;set up reference for
                                    ;c program
        cld                         ;clear direction bit
                                    ;for c program
        fwait                       ;wait for 80287
        call    _my_int_hand        ;call c program
        frstor  env_cop             ;restore environment of
                                    ; the 80287
        cli                         ;turn off interrupts
;
        mov     ax,SEG DGROUP       ;restore previous stack
        mov     es,ax
        mov     ds,es:word ptr save_ss
        mov     ss,es:word ptr save_ss
        mov     sp,es:word ptr save_sp
;
        mov     dx,EOI_PORT         ;load the end of
        mov     ax,EOI_CMD          ;interrupt for the
                                    ;controller chip
        out     dx,al               ;send EOI to the
                                    ;controller chip
        pop     es                  ;pop all the registers
        pop     ds                  ;to return the state
        pop     bp                  ;to what it was before
        pop     di                  ;the interrupt
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        sti
        fwait                       ;restore environment
                                    ; of the 80287
                                    ;let iret enable int's
        iret                        ;return from interrupt
```

```
_int_h_asm      endp
intrp_TEXT      ends
        end
```

```
/*********************************************************
*
*   MY_INT_HAND -- interrupt handler routine
*
*   my_int_hand is called by the interrupt routine
*   int_h_asm.  Interrupt occurs when set number of DMA
*   transmissions has occurred.
*
*   EXTERNAL VARIABLES:
*       x[] = array of (x1, x2, u)
*       comm_bit = communication integer
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/23/88
*
*********************************************************/

#include "loc.h"

void my_int_hand()
{
    extern float x[];
    extern int comm_bit;



    /*      comm_bit = 0
                just received vector, reverse direction
            comm_bit = 1
                just sent back vector, reverse
                direction and update communication
            comm_bit > 1
                error, set to error condition and return */

    if ( comm_bit == 0) {
        send();
        comm_bit = 1;
        x[2] = control( x, UMAX);
    } else if ( comm_bit == 1) {
        receiv();
        comm_bit = 0;
    } else {
      comm_bit = 5;
      }
      /*      turn off transmit/receive signal    */
      auxloff();
}
```

```
/************************************************************
*
*   FI_DRIV -- sets up all the finite element information
*
*   fi_driv()
*
*   fi_driv sets all the necessary finite element
*   information - node table, nodal location, and final
*   time vector.  The final time vector is read in from
*   a user input file.
*
*   AUTHOR:      Donald A. Smith
*   DATE:        6/22/88
*
************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "loc.h"

/*  FUNCTION DEFINITIONS    */
void fortran gridcl( int *, int *, int (*) [3], int *, int *,
                     int *);

void fi_driver()

/* external functions called
fprintf -- c library
node -- generates nodes and nodal position for each
gridlt -- generates nodal table
read_in_nodes -- reads in final time solution vector
*/

{
    int i, j, k, count;
    int nspks, ncir;
    int nelm, maxnelm, nds_read;
    float gridfac, scale;


    extern int ntable[][3];
    extern int ib[];

    /*      number of spokes            */
    nspks = NUMSPKS;

    /*      number of circles           */
    ncir = NUMCIR;
```

```
    /*      scale                         */
    scale = SCALEF;

    /*      grid factor                   */
    gridfac = GRIDF;

    /* read in nodes { #, x, y, tf}      */
    nds_read = read_in_nodes();

    maxnelm = BUF_ELEM;

    /*      generate node table           */
    gridcl( &nspks, &ncir, ntable, &nelm, &maxnelm, ib);
    fprintf(stderr,"#nodes=%d   #elements=%d\n", nds_read, nelm);

    /* shift node table so start with 1 */
    for( i=nelm-1, j=nelm; j > 0; j--, i--) {
        ntable[j][0] = ntable[i][0];
        ntable[j][1] = ntable[i][1];
        ntable[j][2] = ntable[i][2];
    }

}
```

```
/**********************************************************
*
*   READ_IN_NODES -- read in final time vector
*
*   read_in_nodes()
*
*   read_in_nodes reads in the final time vector from a
*   user input file.
*   EXTERNAL VARIABLES:
*       tf[] = final time vector
*   Returns the number of time values read into the
*   the array.
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/22/88
*
**********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "loc.h"

read_in_nodes()

/* external functions called
fprintf -- c library
gets -- c library
fopen -- c library
fclose -- c library
fscanf -- c library
*/

{
    int i, j, count;
    char file_name[40];
    FILE *data_file;
    float dummy;
    extern float tf[];
    extern float x_node[], y_node[];

    do {
        fprintf(stderr,"data file?\n");
        gets(file_name);
        data_file = fopen( file_name,"r");
    } while( data_file == NULL);

    for( i=1; ; i++) {
        if( i > BUF_NODE ) {
            fprintf(stderr,"more nodes than buffer size BUF_NODE\n");
            fprintf(stderr,"aborting read_in\n");
```

```
            break;
        }

        count = fscanf(data_file,"%d",&j);
        if( count == 0 || count == EOF )
            break;
        fscanf(data_file,"%f %f %e",&x_node[j],&y_node[j],&tf[j]);
    }

    if( fclose(data_file) != 0 )
        fprintf(stderr,"%s was not closed\n",file_name);

    return( i-1 );
}
```

```
      SUBROUTINE GRIDCL(NSPOKE, NCIR, NTABLE, NELM, NUMEL, IB)
C     Subroutine to construct the node table for a polar mesh
C     consisting of three node triangles.
C     NSPOKE Integer*4       Input  Number of spokes in mesh.
C     NCIR   Integer*4       Input  Number of circles in mesh.
C     NTABLE Integer*4 Array Output The node table.
C     NELM   Integer*4       Output The total number of elements used.
C     NUMEL  Integer*4       Input  Maximum number of elements.
C     IB     Integer*4 Array Output Array of gradient B.C. types.
      INTEGER*2          NSPOKE, NCIR, NUMEL, NTABLE(3,NUMEL), NELM
      INTEGER*2          IB(NUMEL)
C     Local variables
      INTEGER*4          I, J, N1, N2, N3, N4
      NELM = 0
      DO 500 J = 1, NCIR
          IF(J .EQ. 1) THEN
              DO 100 I = 1, NSPOKE
                  NELM = NELM + 1
                  NTABLE(1,NELM) = 1
                  NTABLE(2,NELM) = I + 1
                  IF( I .EQ. NSPOKE) THEN
                      NTABLE(3,NELM) = 2
                    ELSE
                      NTABLE(3,NELM) = I + 2
                  ENDIF
                  IF(J .EQ. NCIR) THEN
                      IB(NELM) = 2
                    ELSE
                      IB(NELM) = 0
                  ENDIF
100           CONTINUE
            ELSE
              DO 200 I = 1, NSPOKE
                  IF(I .NE. NSPOKE) THEN
                      N1 = (J-1)* NSPOKE + 1 + I
                      N2 = N1 + 1
                      N3 = N2 - NSPOKE
                      N4 = N3 - 1
                      NELM = NELM + 1
                      NTABLE(1,NELM) = N1
                      NTABLE(2,NELM) = N3
                      NTABLE(3,NELM) = N4
                      IB(NELM) = 0
                      NELM = NELM + 1
```

```
                    NTABLE(1,NELM) = N1
                    NTABLE(2,NELM) = N2
                    NTABLE(3,NELM) = N3
                    IF(J .EQ. NCIR) THEN
                        IB(NELM) = 1
                      ELSE
                        IB(NELM) = 0
                    ENDIF
                  ELSE
                    N1 = (J-1) * NSPOKE + 1 + I
                    N2 = N1 - NSPOKE + 1
                    N3 = N2 - NSPOKE
                    N4 = N1 - NSPOKE
                    NELM = NELM + 1
                    NTABLE(1,NELM) = N2
                    NTABLE(2,NELM) = N3
                    NTABLE(3,NELM) = N4
                    IB(NELM) = 0
                    NELM = NELM + 1
                    NTABLE(1,NELM) = N2
                    NTABLE(2,NELM) = N4
                    NTABLE(3,NELM) = N1
                    IF(J .EQ. NCIR) THEN
                        IB(NELM) = 3
                      ELSE
                        IB(NELM) = 0
                    ENDIF
                ENDIF
200           CONTINUE
          ENDIF
500   CONTINUE

      RETURN
      END
```

PRINCIPLE HARDWARE CONFICURATION SUBROUTINES

Five principle routines were used for hardware initialization.

mode0	-	initializes PDMA hardware; large and small model format
mode1	-	sets up DMA controller chip; large and small model format
mode3	-	sets up the interval counters on the PDMA board; large and small model format
mode7	-	installs interrupt routines using BLAISE library support; small model format
mode8	-	installs interrupt routines; large model format

Mode0, 1, 3 can be compiled in the large or small model format. Mode7 is to be compiled in the small model format only. Mode8 is to be compiled in the large model format only. Mode7 uses the support of BLAISE C TOOL library; BLAISE COMPUTING INC., 2560 Ninth Street, Suite 316, Berkeley CA 94710, (415)540-5441.

```
/**************************************************************
*
*
*   MODEO -- initializes the PDMA hardware
*
*   mode0(base_add, dma_level, intp_level)
*
*
*   mode0 is the initialization routine for the PDMA.
*   It checks if the board exists and if the ports are
*   working.
*       base_add   = base address of the board.
*       dma_level  = user selected DMA level; 1,3.
*       intp_level = user selected hardware interrupt
*           level; 2-7.
*   return value is 0 - everything initialized ok
*                   1 - error on initialization
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/16/88
*
**************************************************************/

#include<conio.h>
#include<stdio.h>


mode0(base_add, dma_level, intp_level)

/* external functions called
fprintf -- c library
inpt -- assembly program to do an inport
outpt -- assembly program to do an outport
*/

unsigned base_add;
unsigned dma_level;
unsigned intp_level;

{
    /*      control word for counter 0      */
    unsigned char cw0 = 0x34;
    /*      control word for counter 1      */
    unsigned char cw1 = 0x74;
    /*      control word for counter 2      */
    unsigned char cw2 = 0xb4;
    /*      work variables                  */
    unsigned char retrn, save, work1;
    /*** test if base address is in valid range */
    if (base_add < 0x200 || base_add > 0x3f8 ) {
```

```
    fprintf(stderr,"base address not in range of");
    fprintf(stderr," 200h-3f8h \n");
    fprintf(stderr,"current base address= %u\n",
     base_add);
    return(1);
}
/*** hardware tests */
/** dma test1      */
/*      get current value of dma control reg */
save = retrn = inpt(base_add+2);
/*      set up first test case to output */
work1 = (retrn & 0x03) | (0x4c);
/*      output test case                */
outpt(base_add+2, work1);
/*      read back in                    */
retrn = inpt(base_add+2);
/*      check if what went out = what is read back */
if (retrn != work1) {
    fprintf(stderr,"hardware error, DMA register,");
    fprintf(stderr," test 1\n");
    return(1);
}
/** dma test2     */
/*      set up second test case to output */
retrn = work1 &= 0x03;
/*      output test case                */
outpt(base_add+2, work1);
/*      read case back in               */
retrn = inpt(base_add+2);
/*      check if what went out = what is read back in */
if(retrn != work1) {
    fprintf(stderr,"hardware error, DMA register,");
    fprintf(stderr," test 2\n");
    return(1);
}
/*      restore original contents       */
outpt(base_add+2, save);
/** interrupt test1 */
/*      get current contents            */
save = retrn = inpt(base_add+3);
/*      output test value 54h           */
outpt(base_add+3, 0x54);
/*      read test value back in         */
work1 = inpt(base_add+3);
/*      check if what went out = what is read back in */
if (work1 != 0x54) {
    fprintf(stderr,"hardware error, interrupt");
    fprintf(stderr," register, test 1\n");
    return(1);
}
```

```
    /** interrupt test2 */
    /*      output test value of 0h        */
    outpt(base_add+2,0x0);
    /*      read test value back           */
    workl = inpt(base_add+2);
    /*      check if what went out = what is read back in */
    if(workl != 0x0) {
        fprintf(stderr,"hardware error, interrupt ");
        fprintf(stderr,"register, test 2\n");
        return(1);
    }
    /*      restore to original contents   */
    outpt(base_add+2, save);
    /*      check if DMA level is a valid mode, 1 or 3 */
    if (dma_level !=1 && dma_level !=3 ) {
        fprintf(stderr,"DMA level must be 1 or 3, ");
        fprintf(stderr,"current=%u\n",dma_level);
        return(1);
    }
    /*      check if interrupt level is valid range, 2-7 */
    if (intp_level < 2 && intp_level > 7) {
        fprintf(stderr,"interrupt level out of range 2-7,");
        fprintf(stderr," current=%u\n",intp_level);
        return(1);
    }
    /** set counters to the slowest possible value */
    /*      point to counter 0             */
    outpt(base_add+7, cw0);
    /*      LSB of counter 0               */
    outpt(base_add+4, 0xff);
    /*      MSB of counter 0               */
    outpt(base_add+4, 0xff);
    /*      point to counter 1             */
    outpt(base_add+7, cw1);
    /*      LSB of counter 1               */
    outpt(base_add+5, 0xff);
    /*      MSB of counter 1               */
    outpt(base_add+5, 0xff);
    /*      point to counter 2             */
    outpt(base_add+7, cw2);
    /*      LSB of counter 2               */
    outpt(base_add+6, 0xff);
    /*      MSB of counter 2               */
    outpt(base_add+6, 0xff);
    return(0);
}
```

```
/***********************************************************
*
*
*   MODE1 -- set up DMA controller chip
*
*   model(base_add,dma_level,numb_trans,length,direction,
*         recycle,trans_sourc,segment,offset)
*
*
*   model is used to set up the DMA controller registers and
*   to enable the selected DMA mode.
*           base_add = base address of the PDMA board.
*           dma_level = user selected DMA level, valid
*                levels are 1 and 3.
*           numb_trans = number of bytes/words to be
*                transferred, this is dependent on the
*                mode selected, byte or word transfer.
*           length = what to transfer;
*                0 - byte
*                1 - word
*           direction = direction of the DMA;
*                0 - input from the board to memory.
*                1 - output from memory to the board.
*           recycle = used to allow DMA to recycle- start
*                over at the end by reloading the values
*                it started with;
*                0 - off
*                1 - on
*           trans_sourc = source for Xfer signal;
*                0 - external
*                1 - timer
*           segment = segment the data is located in.
*           offset = offset of starting address of the array
*           of data.
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/16/88
*
***********************************************************/


 model(base_add,dma_level,numb_trans,length,direction,
       recycle,trans_sourc,segment,offset)

/* external functions called
inpt -- assembly program to do an inport
outpt -- assembly program to do an outport
*/

 unsigned base_add;
```

```
unsigned dma_level;
unsigned numb_trans;
unsigned length;
unsigned direction;
unsigned recycle;
unsigned trans_sourc;
unsigned segment;
unsigned offset;

{
    /* data page value                          */
    unsigned page;
    /* data start address                       */
    unsigned trans_add;
    /* PDMA's DMA register                        */
    unsigned pdma_reg;
    /* work variables                           */
    unsigned char work1,work2;
    /* work variable                            */
    unsigned long int work3;
    /* DMA level 1 page register                */
    unsigned char page1 = 0x83;
    /* DMA level 3 page register                */
    unsigned char page3 = 0x82;
    /* 8237 DMA level 1 base register           */
    unsigned char db1 = 0x02;
    /* 8237 DMA level 3 base register           */
    unsigned char db3 = 0x06;
    /* 8237 DMA level 1 byte count register*/
    unsigned char dc1 = 0x03;
    /* 8237 DMA level 3 byte count register*/
    unsigned char dc3 = 0x07;
    /* 8237 DMA mask register                   */
    unsigned char dmask = 0x0a;
    /* 8237 DMA mode register                   */
    unsigned char dmod = 0x0b;

    /* disable selected DMA level before writing
       to its registers                         */
    if (dma_level == 1) {
        outpt(dmask, 0x05);
    } else {
        outpt(dmask, 0x07);
    }

    /*      disable the dma bit in the dma control
            register                            */
    work1 = inpt(base_add+2);
    work1 &= 0x7f;
    outpt(base_add+2, work1);
```

```
/*      setup page register              */
page = segment >> 12;
work3 = (unsigned long int)(segment << 4) +
    (unsigned long int)offset;
/*      trans_add is absolute address    */
trans_add = work3;
/*      increase page if there was a carry on the
        absolute address                 */
page += work3 >> 16;
pdma_reg = inpt(base_add+2);
/*      zero out all but the two aux bits */
pdma_reg &= 0x30;
if (direction) {  /* true=output, false=input */
     if (length)  /* output */
        pdma_reg |= 0x07;     /* word output */
     else
        pdma_reg |= 0x01;     /* byte output */
} else {
     if (length)  /* input */
        pdma_reg |= 0x04;  /*word output, default byte*/
}
if (trans_sourc)  /* true=timer, default external */
     pdma_reg |= 0x08;
if (dma_level == 3)  /* default is for level 1    */
     pdma_reg |= 0x40;
pdma_reg |= 0x80;  /* set enable bit */

/** set up 8237 DMA controller  */
/*      set for single XFER mode        */
work1 = dma_level | 0x40;
if (direction) {
     /*      output = read transfer    */
     work1 |= 0x08;
} else {
     /*      input = write transfer    */
     work1 |= 0x04;
}
if (recycle) {
     /*      auto-initialize on; default off */
     work1 |= 0x10;
}
outpt(dmod, work1);
if (dma_level == 1) {
     /** output page data for level 1  */
     outpt(page1, page);
     if (length) {
          /*      double if byte count is words */
          numb_trans *=2;
     }
     /*      decrease byte count by 1 - DMA default */
```

```
        numb_trans -= 1;
        work2 = numb_trans;
        /*      output low byte of byte count */
        outpt(dc1, work2);
        work2 = numb_trans >> 8;
        /*      output high byte of byte count */
        outpt(dc1, work2);
        work2 = trans_add;
        /*      output low byte of base address */
        outpt(db1, work2);
        work2 = trans_add >> 8;
        /*          output high byte of base address    */
        outpt(db1, work2);
    } else {
        /** output page data for level 3        */
        outpt(page3, page);
        if (length) {
           /*      double if byte count is words */
           numb_trans *=2;
        }
        /*      decrease byte count by 1 - DMA default */
        numb_trans -= 1;
        work2 = numb_trans;
        /*      output low byte of byte count */
        outpt(dc3, work2);
        work2 = numb_trans >> 8;
        /*      output high byte of byte count */
        outpt(dc3, work2);
        work2 = trans_add;
        /*      output low byte of base address */
        outpt(db3, work2);
        work2 = trans_add >> 8;
        /*      output high byte of base address */
        outpt(db3, work2);
    };
    /*      output PDMA register            */
    outpt(base_add+2,pdma_reg);
    /*      enable mask register            */
    outpt(dmask,dma_level);
    return(0);
}
```

```
/**********************************************************
*
*
*   MODE3 -- Set the counters on the PDMA board
*
*   mode3(base_add, div0, div1)
*
*   mode3 is used to set the output frequency for the timers
*   on board the PDMA.
*       base_add = port address of the PDMA.
*       div0     = divider for counter 0.
*       div1     = divider for counter 1.
*   output frequency will be:
*   freq = 10,000,000/(div0*div1)
*
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/16/88
*
**********************************************************/

#include <conio.h>
#include <stdio.h>

mode3(base_add, div0, div1)

/* external functions called
inpt -- assembly program to do an inport
outpt -- assembly program to do an outport
*/

unsigned base_add;
unsigned div0;
unsigned div1;

{
    /*      control word for counter 0      */
    unsigned char cw0 = 0x34;
    /*      control word for counter 1      */
    unsigned char cw1 = 0x74;
    /*      work variable                   */
    unsigned char work1;
    /*      work variable                   */
    unsigned work2;

    /** set up counter # 0                  */
    /*      set counter pointer to counter 0 */
    outpt(base_add+7, cw0);
    /*      strip off upper byte            */
    work1 = (unsigned char)div0;
```

```
    /*      output LSB of divider 0         */
    outpt(base_add+4, work1);
    /*      shift upper byte to lower byte */
    work2 = div0 >> 8;
    /*      strip off upper byte of 0's     */
    work1 = (unsigned char)work2;
    /*      ouput MSB of divider 0          */
    outpt(base_add+4, work1);

    /** set up counter # 1                  */
    /*      set counter pointer to counter 1 */
    outpt(base_add+7, cw1);
    /*      strip off upper byte            */
    work1 = (unsigned char)div1;
    /*      output LSB of divider 1         */
    outpt(base_add+5, work1);
    /*      shift upper byte to lower byte */
    work2 = div1 >> 8;
    /*      strip off upper byte of 0's     */
    work1 = (unsigned char)work2;
    /*      output MSB of divider 1         */
    outpt(base_add+5, work1);
    return (0);
}
```

```
/**********************************************************
*
*   MODE7 -- set up 2 interrupts
*
*   mode7(base_add, intp_level, intp_source, service,
*         action)
*
*    mode7 is used to set up and enable or disable up to 2
*    interrupts.
*       base_add   = port address of the PDMA.
*       intp_level = user defined hardware interrupt number;
*           2-7.
*       intp_source = source of the interrupt;
*              |------   0 = external input, positive slope.
*       int #1 |         1 = external input, negative slope.
*              |         2 = dma terminal count interrupt.
*              |------   3 = PDMA timer interrupt.
*       int #2 --------- 4 = for use in setting up other
*                            interrupts that the PDMA is
*                            not involved in controlling.
*       service = pointer to interrupt service routine.
*       action  = 1 enable the interrupt.
*                 0 disable the interrupt.
*   The enable not only enables the interrupt but also sets
*   up the interrupt vector.  You cannot use this to enable
*   or disable interrupts in a toggle fashion.  Disable
*   reinstalls the previous interrupt vector and also frees
*   the allocated stack.
*
*   Define requirements are:
*       #define STACKSIZE #  where # represents the maximum
*           size of the stack
*       #define NUMSTACKS #  where # represents the maximum
*           number of stacks.
*       This stack number is important mainly if the
*       service routine is recursive.  For each time
*       that it may be called while service the num-
*       stacks must at least equal that, preferable
*       greater by 1 so the program doesn't crash.
*
*   AUTHOR:     Donald A. Smith
*   DATE:       6/16/88
*
***********************************************************/


#include <bisr.h>
#define STACKSIZE   1024
#define NUMSTACKS   2
char *malloc();
```

```
mode7(base_add, intp_level, intp_source, service, action)

/* external functions called
free -- c library
malloc -- c library
inpt -- assembly program to do an inport
outpt -- assembly program to do an outport
isinstall -- blaise library
issetvec -- blaise library
*/

unsigned action;
char *service;
unsigned intp_source;
unsigned base_add;
unsigned intp_level;

{

    /*      ISR control block                */
    static ISRCTRL intr1_ctl,intr2_ctl;
    /*      pointer to ISR's stack           */
    static char *intup1_stack,*intup2_stack;
    /*      work variables                   */
    unsigned char work1, work2;

    /*** install the interrupts              */
    if (action) {

    /*      disable interrupt level          */
    work2 = 0x1;
    /*      shift a 1 to position to mask out int */
    work2 = work2 << (unsigned char)intp_level;
    /*      read current state               */
    work1 = inpt(0x21);
    /*      code to disable selected level */
    work2 |= work1;
    /*      disable interrupt level          */
    outpt(0x21, work2);

    /** setup interrupt service routine      */
    if ( intp_source < 4 ) {   /* install interrupt #1? */
        /*      allocate stack space         */
        intup1_stack = malloc(STACKSIZE*NUMSTACKS);
        /*      install interrupt            */
        isinstal(intp_level+8, service, "iservice",
           &intr1_ctl, intup1_stack, STACKSIZE, NUMSTACKS);

    } else {    /* then install interrupt #2 */
        /*      allocate stack space         */
```

```
     intup2_stack = malloc(STACKSIZE*NUMSTACKS);
     /*       install interrupt          */
     isinstal(intp_level+8, service, "iservice",
        &intr2_ctl, intup2_stack, STACKSIZE, NUMSTACKS);
}

/** set of PDMA control register  */
if ( intp_source < 4 ) {
     /*       get current content to save aux3 */
     work1 = inpt(base_add+3);
     /*       add interrupt level        */
     work2 = intp_level | 0x08;
     /*       shift to left side         */
     work2 = work2 << 0x04;
     if (intp_source == 0x03) {
         /*       adjust to form of control register */
         intp_source++;
     }
     /*       work2 has form except aux3 bit */
     work2 |= intp_source;
     /*       strip all off except aux3 bit  */
     work1 &= 0x08;
     /*       control byte               */
     work2 |= work1;
     /*       output control byte        */
     outpt(base_add+3, work2);
}
/** enable interrupt level */
/*       shift code                      */
work1 = 0x1;
/*       shift bit to correct position   */
work1 = work1 << intp_level;
/*       get current contents            */
work2 = inpt(0x21);
/*       complement mask bit             */
work1= ~(work1);
/*       clear appropriate mask bit      */
work1 &= work2;
/*       set interrupt flag              */
outpt(0x21, work1);
/*       return, no error                */
return(0);

/*** uninstall interrupt                 */
} else {

/** change the interrupt vector table back */
if ( intp_source < 4 ) {
    /*       reset control register      */
    outpt(base_add+3, 0x0);
```

```
    }
    /** disable interrupt level  */
    work2 = 0x1;
    /*      shift a 1 to position to mask out int*/
    work2 = work2 << (unsigned char)intp_level;
    /*      read current state               */
    work1 = inpt(0x21);
    /*      register to disable level        */
    work2 |= work1;
    /*      disable interrupt level          */
    outpt(0x21, work2);

    if ( intp_source < 4 ) {
        /*      install previous vector    */
        issetvec(intp_level+8, &intr1_ctl.prev_vec);
        /*      free up allocated space    */
        free(intup1_stack);
    } else {
        /*      install previous vector    */
        issetvec(intp_level+8, &intr2_ctl.prev_vec);
        /*      free up allocated space    */
        free(intup2_stack);
    }
    return(0);
    }
}
```

```
/**********************************************************
*
*   MODE8 -- set up and install interrupts in a large
*            model program; allows floating point in
*            interrupt routines
*
*   mode8(base_add, intp_level, intp_source, service,
*         action)
*
*   mode8 is used to set up and enable or disable up to 2
*   interrupts.  DOS is bypassed and all the address
*   handling is done by long pointers.  This will only
*   work in the large model format.
*
*        base_add   = port address of the PDMA.
*        intp_level = user defined hardware interrupt number;
*            2-7.
*        intp_source = source of the interrupt;
*               |------   0 = external input, positive slope
*        int #1 |         1 = external input, negative slope
*               |         2 = dma terminal count interrupt
*               |------   3 = PDMA timer interrupt
*        int #2 --------- 4 = for use in setting up other
*                             interrupts that the PDMA is not
*                             involved in controlling
*        service = pointer to interrupt service routine,
*                  for this it is the assembly program
*                  that passes control to the 'C' program
*                  to do the work.
*        action   = 1 enable the interrupt
*                   0 disable the interrupt
*
*   the enable not only enables the interrupt but also sets
*   up the interrupt vector.  You cannot use this to enable
*   or disable it in a toggle fashion.  Disable reinstalls
*   the previous interrupt vector and also frees up the
*   allocated stack.
*
*   AUTHOR:      Donald A. Smith
*   DATE:        6/16/88
*
**********************************************************/

mode8(base_add, intp_level, intp_source, service, action)

unsigned action;
char *service;
unsigned intp_source;
unsigned base_add;
unsigned intp_level;
```

```
{
    /*      previous interrupts             */
    static long unsigned prev_intl, prev_int2;
    /*      pointer to interrupt table      */
    long unsigned *ptr;
    /*      work variables                  */
    unsigned char workl, work2;

    /*      ptr points to interrupt table  */
    ptr = 0x20 + (long unsigned)(intp_level*4);

    /*** install interrupt                 */
    if (action) {

    /*      disable interrupt level         */
    work2 = 0x1;
    /*      shift a 1 to position to mask out int */
    work2 = work2 << (unsigned char)intp_level;
    /*      read current state              */
    workl = inpt(0x21);
    /*      register to disable level       */
    work2 |= workl;
    /*      disable interrupt level         */
    outpt(0x21, work2);

    /** setup interrupt service routine    */
    if ( intp_source < 4 ) { /* install interrupt #1? */
        prev_intl = *ptr;  /* save old interrupt */
        *ptr = service;  /* install new */

    } else {  /* then install interrupt #2 */
        prev_int2 = *ptr;  /* save old interrupt */
        *ptr = service;  /* install new */

    }

    /*** set of PDMA control register      */
    if ( intp_source < 4 ) {
       /*      get current content to save aux3 */
        workl = inpt(base_add+3);
        /*     add interrupt level         */
        work2 = intp_level | 0x08;
        /*     shift to left side          */
        work2 = work2 << 0x04;
        if (intp_source == 0x03) {
             /* adjust to form of control register  */
            intp_source++;
        }
        /*      work2 has form except aux3 bit */
        work2 |= intp_source;
```

```
    /*      strip all off except aux3 bit */
    work1 &= 0x08;
    /*      control byte                  */
    work2 |= work1;
    /*      output control byte           */
    outpt(base_add+3, work2);
}
/** enable interrupt level                */
/*      shift code                        */
work1 = 0x1;
/*      shift bit to correct position     */
work1 = work1 << intp_level;
/*      get current contents              */
work2 = inpt(0x21);
/*      complement mask bit               */
work1= ~(work1);
/*      clear appropriate mask bit        */
work1 &= work2;
/*      set interrupt flag                */
outpt(0x21, work1);
/*      return, no error                  */
return(0);

/*** uninstall interrupt */
} else {

/** change the interrupt vector table back */
if ( intp_source < 4 ) {
    /*      reset control register        */
    outpt(base_add+3, 0x0);
}
/** disable interrupt level               */
work2 = 0x1;
/*      shift a 1 to position to mask out int */
work2 = work2 << (unsigned char)intp_level;
/*      read current state                */
work1 = inpt(0x21);
/*      register to disable level         */
work2 |= work1;
/*      disable interrupt level           */
outpt(0x21, work2);

if ( intp_source < 4 ) {
    /*      install old interrupt         */
    *ptr = prev_int1;

} else {
    /*      install old interrupt         */
    *ptr = prev_int2;
}
```

```
  return(0);
  }
}
```

ASSEMBLY SUPPORT SUBROUTINES

A variety of short assembly subroutines were used to improve execution speed. Two versions of each program are given, large and small model format. The reference frame passed by the C program different for the two formats.

```
auxloff    -   turns Aux 1 line off
auxlon     -   turns Aux 1 line on
aux2off    -   turns Aux 2 line off
aux2on     -   turns Aux 2 line on
inpt       -   get a byte from a port
outpt      -   output a byte to a port
eoi_int    -   send an end-of-interrupt code to the controller
send       -   switch DMA mode 1 to a memory-to-port
                    configuration
receiv     -   switch DMA mode 1 to a port-to-memory
                    configuration
keypush    -   checks if there is something in the keyboard
                    buffer
```

```
;**********************************************************
;*
;*      AUX1OFF -- clear aux1 to a LO state
;*
;*      aux1off() -- small model
;*
;*      aux1off() will clear aux1 on the PDMA board to a LO
;*      state ( 0 ).
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/18/88
;*
;**********************************************************
;
;* external functions calle
;* none
;*
_TEXT   segment byte public 'CODE'  ;define as a code
        assume CS:_TEXT             ;segment
        public _aux1off
_aux1off        proc    near        ;near procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    ax                  ;save state
        push    dx
        mov     dx,302h             ;address aux1 reg
        in      al,dx               ;get current value
        and     al,0efh             ;clear aux1 bit
        out     dx,al               ;output new state
        pop     dx                  ;restore state
        pop     ax
        mov     sp,bp               ;restore stack
        pop     bp
        ret
_aux1off        endp

_TEXT   ends

        end
```

```
;**************************************************************
;*
;*      AUX1OFF -- clear aux1 to a LO state
;*
;*      aux1off() -- large model
;*
;*      aux1off will clear aux1 on the PDMA board to a LO
;*      state ( 0 ).
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/18/88
;*
;**************************************************************
;
;* external functions called
;* none
;*
p1_TEXT segment byte public 'CODE'   ;define as a code
        assume CS:p1_TEXT            ;segment p1_text
        public _aux1off
_aux1off        proc      far        ;far procedure
        push    bp                   ;save current stack
        mov     bp,sp
        push    ax                   ;save state
        push    dx
        mov     dx,302h              ;address of aux1 reg.
        in      al,dx                ;get current value
        and     al,0efh              ;clear aux1 bit
        out     dx,al                ;output new reg.
        pop     dx                   ;restore state
        pop     ax
        mov     sp,bp                ;restore stack
        pop     bp
        ret
_aux1off        endp

p1_TEXT ends

        end
```

```
;***********************************************************
;*
;*      AUX1ON -- set aux1 to a HIGH state
;*
;*      aux1on() -- small model
;*
;*      aux1on will set aux1 on the PDMA board to a HIGH
;*      state ( 1 ).
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/18/88
;*
;***********************************************************
;
;* external functions called
;* none
;*
_TEXT   segment byte public 'CODE'      ;define as a code
        assume CS:_TEXT                 ;segment
        public _aux1on
_aux1on proc    near                    ;far procedure
        push    bp                      ;save current stack
        mov     bp,sp
        push    ax                      ;save current state
        push    dx
        mov     dx,302h                 ;address of aux1 reg
        in      al,dx                   ;get current value
        or      al,10h                  ;set aux1 bit
        out     dx,al                   ;output new reg.
        pop     dx                      ;restore state
        pop     ax
        mov     sp,bp                   ;restore stack
        pop     bp
        ret
_aux1on endp

_TEXT   ends

        end
```

```
;*************************************************************
;*
;*      AUX1ON -- set aux1 to HIGH state
;*
;*      aux1on() -- large model
;*
;*      aux1on will set aux1 on the PDMA board to a HIGH
;*      state ( 1 ).
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/18/88
;*
;*************************************************************
;
;* external functions called
;* none
;*
p2_TEXT segment byte public 'CODE'  ;define as code
        assume CS:p2_TEXT           ;segment p2_text
        public _aux1on
_aux1on proc    far                 ;far procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    ax                  ;save state
        push    dx
        mov     dx,302h             ;address of aux1 reg
        in      al,dx               ;get current value
        or      al,10h              ;set aux1 bit
        out     dx,al               ;output new reg.
        pop     dx                  ;restore state
        pop     ax
        mov     sp,bp               ;restore stack
        pop     bp
        ret
_aux1on endp

p2_TEXT ends

        end
```

```
;**********************************************************
;*
;*      AUX2OFF -- clear aux2 to a LO state
;*
;*      aux2off() -- small model
;*
;*      aux2off will clear aux2 on the PDMA board to a LO
;*      state ( 0 ).
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/18/88
;*
;**********************************************************
;
;* external functions called
;* none
;*
_TEXT   segment byte public 'CODE'  ;define as a code
        assume CS:_TEXT             ;segment
        public _aux2off
_aux2off        proc    near        ;near procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    ax                  ;save current state
        push    dx
        mov     dx,302h             ;address of aux2 reg.
        in      al,dx               ;get current value
        and     al,0dfh             ;clear aux2 bit
        out     dx,al               ;ouput new reg.
        pop     dx                  ;restore state
        pop     ax
        mov     sp,bp               ;restore stack
        pop     bp
        ret
_aux2off        endp

_TEXT   ends

        end
```

```
;*******************************************************
;*
;*      AUX2OFF -- set aux2 to a LO state
;*
;*      aux2off() -- large model
;*
;*      aux2off will clear aux2 on the PDMA board to a LO
;*      state ( 0 ).
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/18/88
;*
;*******************************************************
;
;* external functions called
;* none
;*
p3_TEXT segment byte public 'CODE'   ;define as a code
        assume CS:p3_TEXT            ;segment p3_text
        public _aux2off
_aux2off        proc    far          ;far procedure
        push    bp                   ;save current stack
        mov     bp,sp
        push    ax                   ;save current state
        push    dx
        mov     dx,302h              ;address of aux2 reg.
        in      al,dx                ;get current value
        and     al,0dfh              ;clear aux2 bit
        out     dx,al                ;output new reg.
        pop     dx                   ;restore state
        pop     ax
        mov     sp,bp                ;restore stack
        pop     bp
        ret
_aux2off        endp

p3_TEXT ends

        end
```

```
;*****************************************************************
;*
;*    AUX2ON -- set aux2 to a HIGH state
;*
;*    aux2on() -- small model
;*
;*    aux2on will set aux2 on the PDMA board to a HIGH
;*    state ( 1 ).
;*
;*    AUTHOR:         Donald A. Smith
;*    DATE:           6/18/88
;*
;*****************************************************************
;
;* external functions called
;* none
;*
_TEXT segment byte public 'CODE'  ;define as a code
      assume    CS:_TEXT               ;segment
      public    _aux2on
_aux2on    proc near                   ;near procedure
      push bp                     ;save current stack
      mov  bp,sp
      push ax                     ;save current state
      push dx
      mov  dx,302h                ;address of aux2 reg.
      in   al,dx                  ;get current value
      or   al,20h                 ;set aux2 bit
      out  dx,al                  ;output new reg.
      pop  dx                     ;restore state
      pop  ax
      mov  sp,bp                  ;restore stack
      pop  bp
      ret
_aux2on    endp

_TEXT ends

      end
```

```
;*************************************************************
;*
;*    AUX2ON -- set aux2 to a HIGH state
;*
;*    aux2on() -- large model
;*
;*    aux2on will set aux2 on the PDMA board to a HIGH
;*    state ( 1 ).
;*
;*    AUTHOR:         Donald A. Smith
;*    DATE:           6/18/88
;*
;*************************************************************
;
;* extenal functions called
;* none
;*
p4_TEXT    segment byte public 'CODE'  ;define as a code
      assume    CS:p4_TEXT             ;segment p4_text
      public    _aux2on
_aux2on    proc far                    ;far procedure
      push bp                     ;save current stack
      mov  bp,sp
      push ax                     ;save current state
      push dx
      mov  dx,302h                ;address of aux2 reg.
      in   al,dx                  ;get current value
      or   al,20h                 ;set aux2 bit
      out  dx,al                  ;output new reg.
      pop  dx                     ;restore state
      pop  ax
      mov  sp,bp                  ;restore stack
      pop  bp
      ret
_aux2on    endp

p4_TEXT    ends

      end
```

```
;**********************************************************
;*
;*      INPT -- get a value from a port
;*
;*      int inpt(port#) -- small model
;*
;*      inpt gets a value from a port.
;*           port# = integer value of port number
;*      returns the value as an integer
;*
;*      AUTHOR:          Donald A. Smith
;*      DATE:            6/18/88
;*
;**********************************************************
;
;* external functions called
;* none (replaces c version 'inp(port#)' )
;*
_TEXT   segment byte public 'CODE'  ;define as code
        assume  CS:_TEXT            ;segment
        public  _inpt
_inpt   proc near                   ;far procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    dx                  ;save state
        mov     ax,0                ;clear ax reg.
        mov     dx,[bp+4]           ;get port#
        in      al,dx               ;inport the value
        pop     dx                  ;return in ax reg.
        mov     sp,bp               ;restore state
        pop     bp                  ;restore stack
        ret
_inpt   endp

_TEXT   ends

        end
```

```
;***********************************************************
;*
;*      INPT -- get a value from a port
;*
;*      int inpt(port#) -- large model
;*
;*      inpt gets a value from a port.
;*           port# = integer value of port number
;*      returns the value as an integer
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/18/88
;*
;***********************************************************
;
;* external functions called
;* none (replaces c version 'inp(port#)' )
;*
p8_TEXT segment byte public 'CODE'   ;define as code
        assume  CS:p8_TEXT           ;segment p8_text
        public  _inpt
_inpt   proc    far                  ;far procedure
        push    bp                   ;save current stack
        mov     bp,sp
        push    dx                   ;save state
        mov     ax,0                 ;clear ax reg.
        mov     dx,[bp+6]            ;get port#
        in      al,dx                ;inport the value
                                     ;return in ax reg.
        pop     dx                   ;restore state
        mov     sp,bp                ;restore stack
        pop     bp
        ret
_inpt   endp

p8_TEXT   ends

        end
```

```
;*************************************************************
;*
;*      OUTPT -- output a value to a port
;*
;*      outpt(port#, value) -- small model
;*
;*      outpt -- out puts a value to a port
;*           port# = unsigned integer port number
;*           value = integer value to output ( only the
;*                   lower byte will be output)
;*
;*      AUTHOR:          Donald A. Smith
;*      DATE:            6/18/88
;*
;*************************************************************
;
;* external functions called
;* none  (replaces c version of outp(port#, value)  )
;*
_TEXT   segment byte public 'CODE'  ;define as a code
        assume  CS:_TEXT            ;segment
        public  _outpt
_outpt  proc near                   ;far procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    ax                  ;save state
        push    dx
        mov     dx,[bp+4]           ;get port number
        mov     ax,[bp+6]           ;get output value
        out     dx,al               ;output value
        pop     dx                  ;restore state
        pop     ax
        mov     sp,bp               ;restore stack
        pop     bp
        ret
_outpt  endp

_TEXT   ends

        end
```

```
;*****************************************************************
;*
;*      OUTPT -- output a value to a port
;*
;*      outpt(port#, value) -- large model
;*
;*      outpt -- out puts a value to a port
;*           port# = unsigned integer port number
;*           value = integer value to output ( only the
;*                   lower byte will be output)
;*
;*      AUTHOR:          Donald A. Smith
;*      DATE:            6/18/88
;*
;*****************************************************************
;
;* external functions called
;* none  (replaces c version of outp(port#, value)  )
;*
p10_TEXT        segment byte public 'CODE';define as a code
        assume  CS:p10_TEXT             ;segment p10_text
        public  _outpt
_outpt  proc    far                     ;far procedure
        push    bp                      ;save current stack
        mov     bp,sp
        push    ax                      ;save state
        push    dx
        mov     dx,[bp+6]               ;get port number
        mov     ax,[bp+8]               ;get output value
        out     dx,al                   ;output value
        pop     dx                      ;restore state
        pop     ax
        mov     sp,bp                   ;restore stack
        pop     bp
        ret
_outpt  endp

p10_TEXT   ends

        end
```

```
;*********************************************************
;*
;*      EOI_INT -- send an end-of-interrupt (EOI)
;*
;*      eoi_int() -- small model
;*
;*      eoi_int will send an end-of-interrupt to the 8259
;*      interrupt controller chip indicating a service to
;*      an interrupt.
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/16/88
;*
;*********************************************************
;
;* external functions called
;* none
;*
_TEXT   segment byte public 'CODE'  ;define as a code
        assume  CS:_TEXT            ;segment p7_text
        public  _eoi_int
_eoi_int        proc    near        ;far procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    ax                  ;save current state
        mov     al,20h              ;EOI signal
        out     20h,al              ;send EOI to 8259
        pop     ax                  ;restore state
        mov     sp,bp               ;restore stack
        pop     bp
        ret
_eoi_int        endp

_TEXT   ends

        end
```

```
;**********************************************************
;*
;*      EOI_INT -- send an end-of-interrupt (EOI)
;*
;*      eoi_int() -- large model
;*
;*      eoi_int will send an end-of-interrupt to the 8259
;*      interrupt controller chip indicating a service to
;*      an interrupt.
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/16/88
;*
;**********************************************************
;
;* external functions called
;* none
;*
p7_TEXT segment byte public 'CODE'  ;define as a code
        assume  CS:p7_TEXT          ;segment p7_text
        public  _eoi_int
_eoi_int        proc    far         ;far procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    ax                  ;save current state
        mov     al,20h              ;EOI signal
        out     20h,al              ;send EOI to 8259
        pop     ax                  ;restore state
        mov     sp,bp               ;restore stack
        pop     bp
        ret
_eoi_int        endp

p7_TEXT ends

        end
```

```
;***********************************************************
;*
;*      SEND -- switch DMA mode 1 to memory-to-port
;*
;*      send() -- small model
;*
;*      send changes the registers on the PDMA board
;*      and the DMA controller chip to a memory-to-port
;*      status.  Warning -- even though is disables DMA
;*      first, this should be used with caution.
;*
;*      AUTHOR:         Donald A. Smith
;*      DATA:           6/21/88
;*
;***********************************************************
;
;* external functions called
;* none
;*
_TEXT   segment byte public 'CODE'  ;define as code
        assume CS:_TEXT             ;segement
        public _send
_send   proc near                   ;far procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    ax                  ;save current state
        push    dx
        mov     al,05h              ;mask DMA level 1
        out     0ah,al              ;output mask to reg.
        mov     al,59h              ;code for mode reg. to
        out     0bh,al              ;change directions
        mov     dx,302h             ;address of PDMA reg.
        in      al,dx               ;get current value
        and     al,30h              ;set direction
        or      al,87h
        out     dx,al               ;output new value
        mov     al,01h              ;enable DMA, level 1
        out     0ah,al
        pop     dx                  ;restore state
        pop     ax
        mov     sp,bp               ;restore stack
        pop     bp
        ret
_send   endp

_TEXT   ends

        end
```

```
;*************************************************************
;*
;*      SEND -- switch DMA mode 1 to memory-to-port
;*
;*      send() -- large model
;*
;*      send changes the registers on the PDMA board
;*      and the DMA controller chip to a memory-to-port
;*      status.  Warning -- even though is disables DMA
;*      first, this should be used with caution.
;*
;*      AUTHOR:         Donald A. Smith
;*      DATA:           6/21/88
;*
;*************************************************************
;
;* external functions called
;* none
;*
p12_TEXT        segment byte public 'CODE' ;define as code
        assume CS:p12_TEXT          ;segement p12_TEXT
        public _send
_send   proc    far                 ;far procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    ax                  ;save current state
        push    dx
        mov     al,05h              ;mask DMA level 1
        out     0ah,al              ;output mask to reg.
        mov     al,59h              ;code for mode reg. to
        out     0bh,al              ;change directions
        mov     dx,302h             ;address of PDMA reg.
        in      al,dx               ;get current value
        and     al,30h              ;set direction
        or      al,87h
        out     dx,al               ;output new value
        mov     al,01h              ;enable DMA, level 1
        out     0ah,al
        pop     dx                  ;restore state
        pop     ax
        mov     sp,bp               ;restore stack
        pop     bp
        ret
_send   endp

p12_TEXT        ends

        end
```

```
;**********************************************************
;*
;*      RECEIV -- switch active DMA to  port-to-memory
;*
;*      receiv() -- small model
;*
;*      receiv() changes the registers on the PDMA board
;*      and the DMA controller chip to a port-to-memory
;*      status.  Warning -- even though it disables DMA
;*      first this should be used with caution.
;*      ASSUMPTIONS:
;*              DMA level 1 is being used
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/21/88
;*
;**********************************************************
;
;* external functions called
;* none
;*
_TEXT   segment byte public 'CODE'   ;define as code
        assume CS:_TEXT              ;segment
        public _receiv
_receiv proc near                    ;far procedure
        push    bp                   ;save current stack
        mov     bp,sp
        push    ax                   ;save state
        push    dx
        mov     al,05h               ;mask DMA level 1
        out     0ah,al               ;output mask to reg.
        mov     al,55h               ;code for mode reg. to
        out     0bh,al               ;change directions
        mov     dx,302h              ;address of PDMA reg.
        in      al,dx                ;get current value
        and     al,30h               ;clear direction
        or      al,84h
        out     dx,al                ;output new value
        mov     al,01h               ;enable DMA, level 1
        out     0ah,al
        pop     dx                   ;restore state
        pop     ax
        mov     sp,bp                ;restore stack
        pop     bp
        ret
_receiv endp


_TEXT   ends
```

```
end
```

```
;*****************************************************************
;*
;*      RECEIV -- switch DMA mode 1 to  port-to-memory
;*
;*      receiv() -- large model
;*
;*      receiv() changes the registers on the PDMA board
;*      and the DMA controller chip to a port-to-memory
;*      status.  Warning -- even though it disables DMA
;*      first this should be used with caution.
;*
;*      AUTHOR:         Donald A. Smith
;*      DATE:           6/21/88
;*
;*****************************************************************
;
;* external functions called
;* none
;*
pll_TEXT        segment byte public 'CODE' ;define as code
        assume CS:pll_TEXT          ;segment pll_TEXT
        public _receiv
_receiv proc    far                 ;far procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    ax                  ;save state
        push    dx
        mov     al,05h              ;mask DMA level 1
        out     0ah,al              ;output mask to reg.
        mov     al,55h              ;code for mode reg. to
        out     0bh,al              ;change directions
        mov     dx,302h             ;address of PDMA reg.
        in      al,dx               ;get current value
        and     al,30h              ;clear direction
        or      al,84h
        out     dx,al               ;output new value
        mov     al,01h              ;enable DMA, level 1
        out     0ah,al
        pop     dx                  ;restore state
        pop     ax
        mov     sp,bp               ;restore stack
        pop     bp
        ret
_receiv endp


pll_TEXT        ends

        end
```

```
;***********************************************************
;*
;*      KEY_PUSH -- check if there is something in the
;*      keyboard buffer
;*
;*      int key_push() -- small model
;*
;*      key_push checks the keyboard buffer for a keystroke.
;*      returns a nonzero value if a key has been pressed
;*
;*      AUTHOR:         Donald A. Smith
;*      DATA:           6/18/88
;*
;***********************************************************
;
;* external functions called
;* none
;*
_TEXT   segment byte public 'CODE'      ;define as code
        assume  CS:_TEXT                ;segment _text
        public  _key_push
_key_push       proc    near            ;near procedure
        push    bp
        mov     bp,sp
        push    es                      ;save used register
        mov     ax,0h                   ;zero out ax reg.
        mov     es,ax                   ;zero out es reg.
        mov     al,es:[41ah]            ;get head pointer
        mov     ah,es:[41ch]            ;get tail pointer
        cmp     ah,al                   ;compare head & tail
        jne     SOMETHING               ;if not equal jump
        mov     ax,0h                   ;return a 0
        jmp     WE_DONE                 ;exit
SOMETHING:
        mov     ax,01h                  ;return a 1
WE_DONE:
        pop     es                      ;restore state
        mov     sp,bp                   ;restore stack
        pop     bp
        ret
_key_push       endp

_TEXT   ends

        end
```

```
;**********************************************************
;*
;*      KEY_PUSH -- check if there is something in the
;*      keyboard buffer
;*
;*      int key_push() -- large model
;*
;*      key_push checks the keyboard buffer for a keystroke.
;*      returns a nonzero value if a key has been pressed
;*
;*      AUTHOR:         Donald A. Smith
;*      DATA:           6/18/88
;*
;**********************************************************
;
;* external functions called
;* none
;*
p9_TEXT segment byte public 'CODE'  ;define as a code
        assume  CS:p9_TEXT          ;segment p9_text
        public  _key_push
_key_push       proc    far         ;far procedure
        push    bp                  ;save current stack
        mov     bp,sp
        push    es                  ;save current state
        mov     ax,0h               ;zero out ax reg.
        mov     es,ax               ;zero out es reg.
        mov     al,es:[41ah]        ;get head pointer
        mov     ah,es:[41ch]        ;get tail pointer
        cmp     ah,al               ;compare head & tail
        jne     SOMETHING           ;if not equal jump
        mov     ax,0h               ;return a 0
        jmp     WE_DONE             ;exit
SOMETHING:
        mov     ax,01h              ;return a 1
WE_DONE:
        pop     es                  ;restore state
        mov     sp,bp               ;restore stack
        pop     bp
        ret
_key_push       endp

p9_TEXT ends

        end
```

REFERENCES

1. Bushaw, A.E., "Optimal Discontinuous Forcing Terms," IN: Contributions to the Theory of Nonlinear Oscillations, Vol. IV, pp. 29-52, Princeton University Press, 1958.

2. Bellman, R., Glicksberg, I., and Cross, O., "On the 'Bang-Bang' Control Problem," Quarterly of Applied Mathematics, Vol. 14, pp. 11-18, 1956.

3. Pontryagin, L.S., Boltyanski, V.C., Gamrelidze, R.V., and Mischenko, E.F., The Mathematical Theory of Optimal Processes, New York: Interscience, 1962.

4. LaSalle, J.P., "The Time Optimal Control Problem," IN: Contributions to the Theory of Nonlinear Oscillations, Vol. V, pp. 1-24, Princeton University Press, 1960.

5. Oldenburger, R., and Thompson, C., "Introduction to Time Optimal Control of Stationary Linear Systems," Automatica, Vol. 1, pp. 177-205, 1963.

6. Shetty, A., "A Solution Technique for the Minimum-Time Control Problem of an R-Theta Manipulator," M.S. Thesis in Mechanical Engineering, Kansas State University, 1987.

7. Subrahmanyam, M.B., "A Computational Method of the Solution of Time-Optimal Control Problems by Newton's Method," International Journal of Control, Vol. 44, No. 5, pp. 1233-1243, 1986.

8. Knudsen, H.K., "An Iterative Procedure for Computing Time-Optimal Controls," IEEE Transactions on Automatic Control, Vol. AC-9, No. 1, pp. 23-30, January, 1964.

9. Lastman, C.L., "A Shooting Method for Solving Two-Point Boundary-Value Problems Arising from Non-Singular Bang-Bang Optimal Control Problems," International Journal of Control, Vol. 27, No. 4, pp. 513-524, 1978.

10. Lasdon, L.S., Mitter, S.K., and Warren, A.D., "The Conjugate Cradient Method for Optimal Control Problems," IEEE Transactions on Automatic Control, Vol. AC-15, No. 2, pp. 132-138, 1967.

11. Lewine, R.N., and Thorp, J.S., "Computation of Time-Optimal Controls, Using a Second-Variation Descent Search," IEEE Transactions on Automatic Control, Vol. AC-15, No. 3, pp. 358-362, June, 1970.

12. Kahn, M.E., and Roth, B., "The Near-Minimum-Time Control of Open-Loop Articulated Kinematic Chains," Transactions of ASME, Journal of Dynamic Systems, Measurement, and Control, pp. 164-172, September, 1971.

13. Larson, V.H., "Minimum Time Control by Time Interval Optimization," International Journal of Control, Vol. 7, No. 4, pp. 381-394, 1968.

14. Smith, F.B. Jr., "Time Optimal Control of High Order Systems," IRE Transactions on Automatic Control, Vol. AC-6, pp. 16-21, February, 1961.

15. Yastreboff, M., "Synthesis of Time-Optimal Control by Time Interval Adjustment," IEEE Transactions on Automatic Control, Vol. AC-14, No. 6, pp. 707-710, December, 1969.

16. Davison, E.J., and Monro, D.M., "A Computational Technique for Finding Time Optimal Controls of Nonlinear Time-Varying Systems," Proceedings of the Joint Automatic Controls Conference, pp. 270-280, 1969.

17. Wen, J.T., and Desrochers, A.A., "An Algorithm for Obtaining Bang-Bang Control Laws," Transactions of ASME, Journal of Dynamic Systems, Measurement, and Control, Vol. 109, No. 2, pp. 171-175, June, 1987.

18. Niemann, D.D., "Determination of Bang-Bang Controls for Large Nonlinear Systems," M.S. Thesis in Mechanical Engineering, Kansas State University, 1988.

19. Lee, E.B., and Markus, L., Foundations of Optimal Control Theory, NY: John Wiley, 1967.

20. Athans, M., and Falb, P.L., Optimal Control, McGraw-Hill Inc., 1966.

21. Ryan, E.P., "Time-Optimal Control of Certain Plants with Positive Real Eigenvalues," International Journal of Control, Vol. 23, No. 6, pp. 775-783, 1976.

22. Ryan, E.P., "Minimum Time Isochronal Surfaces for Certain Third-Order Systems", International Journal of Control, Vol. 26, No. 3, pp. 421-433, 1977.

23. Rajendran, K., "A Continuum Approach to Minimum Time Control," M.S. Thesis in Mechanical Engineering, Kansas State University, 1988.

24. Luh, J.Y., and Shafran, J.S., "An Approximate Minimal Time Closed-Loop Controller for Processes with Bounded Control Amplitudes and Rates," Proceedings of the Joint Automatic Control Conference, Boulder, Colorado, pp. 623-632, 1969.

25 Smith, F.B. Jr., "Design of Quasi-Optimal Minimum-Time Controllers," IEEE Transactions on Automatic Control, Vol. AC-11, No. 1, January, 1966.

26. Bryson, A.E. Jr., and Ho, Y.C., Applied Optimal Control, John Wiley and Sons, 1975.

27. White, W.N. Jr., and Rajendran, K., "A Continuum Approach to Mimimum Time",Proceedings of the 1988 American Control Conference, Atlanta, Georgia, pp. 2050-2054.

28. Huebren, K.H., and Thorton, E.A., The Finite Element Method for Engineers, John Wiley and Sons, 1982.

29. PDMA-16 Manual, MetraByte Corporation, 440 Standish Boulevard, Taunton, Mass. 02780.

30. Hunt, W.J., The C Tool Box, Addison-Wesely Publishing Company Inc., 5th ed., 1987.

31. Horowitz, P., and Hill, W.O., The Art of Electronics, Cambridge University Press, 1986.

32. Kerningham, B.W., and Ritchie, D.M., The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

33. Chesely, H.R., and Waite, M., Supercharging C with Assembly Language, Addision-Wesely Publishing Company Inc, 1987.

34. The TTL Data Book, Texas Instruments Inc., Vol. 2, 1985.

35. INTEL Microsystems Components Handbook, INTEL Inc., Vol. 1, 1986.

A FEASIBILITY ASSESSMENT OF A FINITE ELEMENT
REAL TIME, TIME OPTIMAL CONTROLLER

by

DONALD ALLAN SMITH

B.S., Kansas State University, 1985

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

MECHANICAL ENGINEERING

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

ABSTRACT

This paper is an investigation into using minimum time isochrones in a phase plane to control a double integrator system. The desired control is the time optimal bang-bang control. The isochrones provide an approximate means to determine the control as a function of the state variables.

The isochrones are approximated by a discrete final time grid. The final time solution is determined using finite element techniques. The finite elements modeled the system's phase plane.

Two grids are used to determine the isochrones, one Cartisian, the other polar in shape. To test the two grids, a plant-controller system is used based upon two personal computers. One computer works as the controller while the other simulates the double integrator system.

The control can be calculated from the finite element grid in 0.02 seconds. The finite element grids were solved off-line and used by the controller during the real-time simulation. The control generated using the square grid chattered in the region close to the origin. The polar grid provided a good control throughout the phase plane. The simulations conducted using this grid produced elapsed time values close to optimal values.